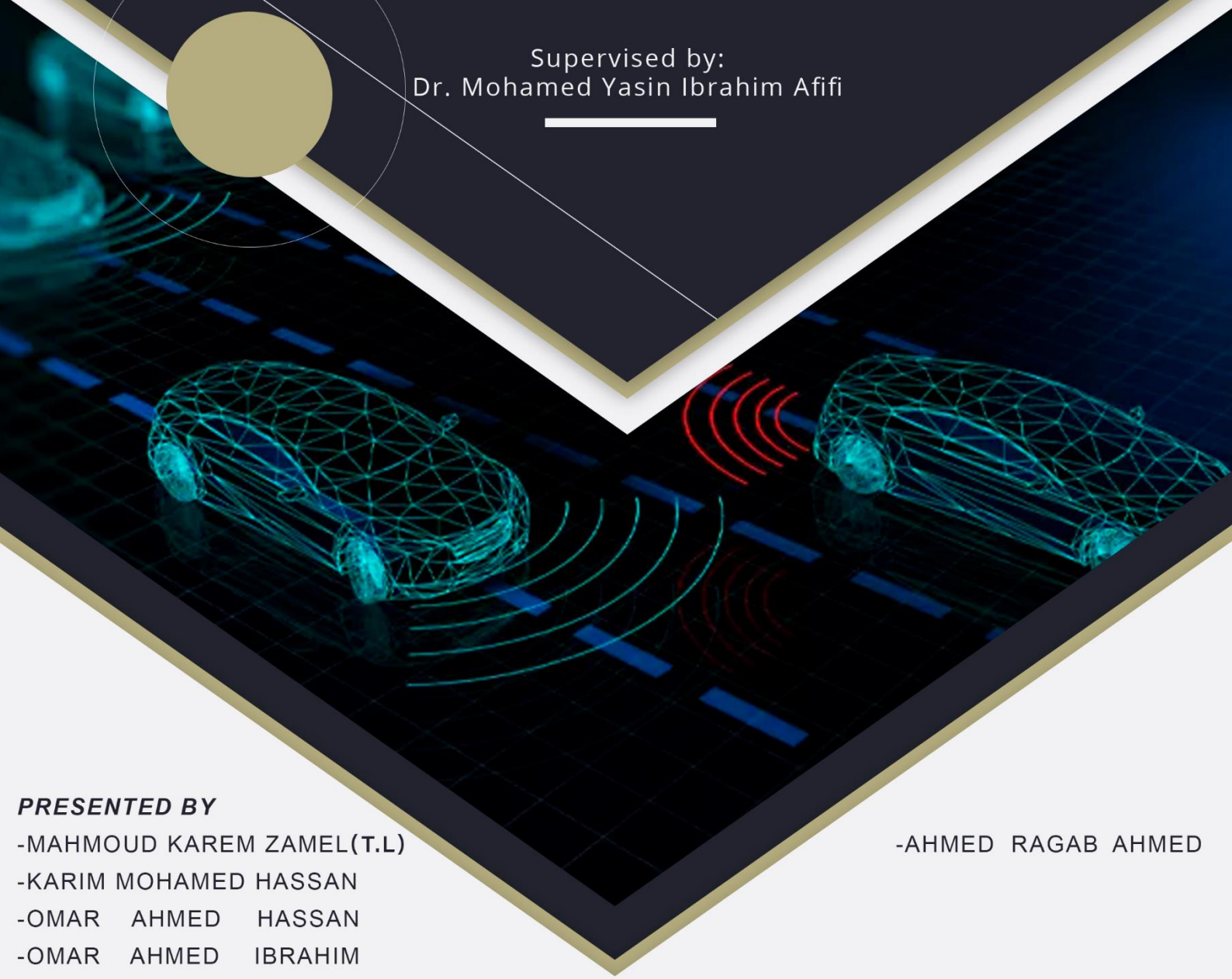
Al-Azhar University
Faculty of Engineering
Electronics & Commmunication
Department

Valeo

B.Sc.
GRADUATION PROJECT

V2V COLLISION AVOIDANCE

ELECTRONICS & COMMUNICATION



Supervised by:
Dr. Mohamed Yasin Ibrahim Afifi

PRESENTED BY

-MAHMOUD KAREM ZAMEL(**T.L**)
-KARIM MOHAMED HASSAN
-OMAR AHMED HASSAN
-OMAR AHMED IBRAHIM
-MOHAMED YASSER KAMAL
-MOHAMED JAMEL ELZEND
-AHMED RAGAB AHMED

Acknowledgement

First things first we would like to express our sincere gratitude for everyone who helped us during the graduation project, starting with endless thanks for our supervisor, **Dr. Mohamed Yasin Ibrahim** who didn't keep any effort in encouraging us to do a great job, providing our group with valuable information and advice to be better each time. Thanks for the continuous support and kind communication which had a great effect regarding to feel interesting about what we are working on.

Thanks are extended to **Valeo** with the help of **Eng. Raghda Samir** and **Eng. Ahmed Hassan** for their time, endless support and their huge effort in contacting and providing us with all what we need regarding information to help us overcome some obstacles we faced.

Also, we cannot forget our college **Faculty of Engineering - Al-Azhar University** forgiving us a good education, help and chances to achieve our goals.

Finally, we will never find enough words to express the gratitude and appreciation that we owe to our families for their constant support, encouragement, never-ending patience as well as their love that raised us up again when we got weary and without them, we doubt that we would be in this place today.

Abstract

Vehicle-to-vehicle (V2V) communication is a promising technology for improving road safety by enabling vehicles to exchange information about their speed, position, and direction in real-time. In this graduation project, we propose a V2V collision avoidance advanced driver assistance system (ADAS) that can reduce the risk of accidents by alerting drivers of potential collisions and assisting them in taking corrective actions.

The proposed system consists of two main components: a communication module that enables vehicles to exchange data via wireless communication, and a collision avoidance module that analyzes the received data and generates warnings and/or corrective actions if a collision is imminent. The system uses various sensors to detect obstacles and other vehicles, and predict collision risks.

To evaluate the performance of the proposed system, we conducted a series of experiments using simulation projectiles. The results show that the V2V collision avoidance ADAS system can effectively reduce the number of collisions and improve driving safety. Moreover, we analyzed the system's scalability, reliability, and interoperability, and we discussed the potential challenges and opportunities for future research in this area.

Overall, this graduation project demonstrates the feasibility and effectiveness of V2V collision avoidance ADAS systems and highlights their potential for improving road safety and reducing accidents.

Contents

Contents 4
List of figures: 10
List of tables 16
Table of an abbreviations 16
Introduction 20
0.1 V2V Communication 20
0.1.1 What is V2V 20
0.1.2 V2V Market 21
0.1.3 Competition Landscape 21
0.1.4 Standardized V2V Protocol 22
0.2 Project Objective 23
0.3 Motivation 23
0.4 Project Description 24
0.5 Research Methodology, Implementation and Validation 28
0.6 Development Plan 29
0.6.1 Project Time Line 29
0.6.2 Global design Document (GDD) 30
0.7 Project Architecture 31
Main ECU Raspberry pi 35
1.1 Introduction 35
1.2 Why Raspberry pi 37
1.3 Raspberry pi Versions 37
1.4 Raspberry pi Hardware 40
1.4.1 Specifications Pin Diagram and Description 40
1.4.2 Hardware Components Of Raspberry pi 40
1.4.3 Layout of The GPIO header 43
1.5 Raspberry pi OS 44

Communicatio ECU(ESP32) 45
2.1 ESP32 45
2.1.a ESP-NOW 46
2.2 Hardware ESP32 49
2.2.a ESP32 Peripherals and I/O 50
2.3 How to Connect the ESP32 With Raspberry Pi 51
2.4 Raspberry Pi with MC Using UART 53
2.4.a Hardware Connection of Raspberry Pi with MC Using UART 54
Sensors&Actuators 55
3.1 LCD 55
3.1.1 LCD Theory of work 55
3.1.2 About the module 55
3.1.3 LCD Pin Configuration 55
3.1.4 Use of LCD in out project 57
3.1.5 Optimization 57
3.2 ULTRASONIC SENSOR 58
3.2.1 Ultrasonic theory of work 58
3.2.2 About the Module 58
3.2.3 Working Method 60
3.2.4 Use of Ultrasonic in our project 61
3.2.5 Optimization 62
3.2.6 Ultrasonic Flowchart 63
3.3 DC MOTOR 64
3.3.1 About the module 64
3.3.2 About the H-Bridge 65
3.3.3 Working Method 66
3.4 HMC5883L Sensor 68
3.4.1 About the module 69
3.4.2 HMC5883L theory of work 69
3.4.3 Use of HMC5883L in our project 70
3.5 BUZZER 71

3.5.1 Buzzer Pin Configuration 71
3.5.2 Specifications 71
3.5.3 Working principle 72
3.5.4 Buzzer Circuit Diagram 73
ESP NOW 75
4.1 Introduction 75
4.2 ESP NOW 76
4.2.1 ESP NOW Networking Modes 76
4.2.1.a Initiators and Responders 76
4.2.1.b One-Way Communication 77
4.2.1.c One Initiator & Multiple Responders 77
4.2.1.d One Responder & Multiple Initiators 77
4.2.1.f Two-Way Communications 77
4.2.1.g Two-Way Networking 78
4.2.2 MAC Addresses 79
4.2.3 MAC Address Sketch 80
4.2.3.a Running the MAC Sketch 80
4.3 Coding for ESP-NOW 81
4.3.1 Callbacks 81
4.3.2 Coding for ESP-NOW – Sending 82
4.3.3 Coding for ESP-NOW – Receiving 82
4.4 More Information About ESP NOW 83
4.4.1 Frame Format 83
4.4.2 Security 84
4.4.3 Initialization and De-initialization 85
4.4.4 Add Paired Device 85
4.4.5 Send ESP-NOW Data 86
4.4.6 Receiving ESP-NOW Data 87
4.4.7 Config ESP-NOW Rate 87
4.4.8 Config ESP-NOW Power-saving Parameter 87
UART 88

5.1 Introduction 88
5.2 UART Interfacing 89
5.3 Data Transmission 91
5.3.1 Start Bit 91
5.3.2 Data Frame 92
5.3.3 Parity 92
5.3.4 Stop Bits 92
5.4 Steps of UART Transmission 93
5.5 UART Operations 95
5.6 Advantages 98
5.7 Disadvantages 98
5.8 Use Cases 98
Bluetooth 99
6.1 Introduction 99
6.2 How Bluetooth Work 100
6.3 Bluetooth Profiles 101
6.4 Bluetooth Versions 104
6.5 Bluetooth architecture of ESP32 105
6.5.1 Bluetooth Application Structure 105
6.5.2 Selection of the HCI Interfaces 107
6.5.3 Bluetooth Operating Environment 109
Communication between ESP32 and Raspberry Pi 110
7.1 The Communication Between The 2 ESP32 110
7.1.1 ESP32 in Our System 110
7.1.2 ESP NOW 110
7.1.3 Broadcast Communication Mode 112
7.2 The Communication Between the Raspberry Pi and the ESP32 113
7.2.1 UART in ESP32 113
7.2.2 UART in Raspberry Pi 113
7.3 The Shape of The Message 114
EEBL&FCW 116

8.1 EEBL 116
8.1.1 Description Of Alternatives 116
8.2 FCW 117
8.2.1 Description Of Alternatives 117
8.3 EEBL&FCW 118
8.3.1 Description Of Alternatives 118
8.3.2 Main algorithm 119
8.3.3 ID_algorithm 120
8.3.4 EEBL&FCW Layered Architecture 121
8.3.5 Systems APIs Description 122
BSW-DNPW 124
9.1 BSW-DNPW 124
9.1.1 System Description 124
9.1.2 Main algorithm 125
9.1.3 ID Algorithm 126
9.2 Layered Architecture 126
9.3 APIs Description 127
IMA 129
10.1 IMA 129
10.1.a System Description 129
10.1.b Main Algorithm 130
10.1.c ID Algorithm 132
10.1.d IMA Layered Architecture 133
10.1.e IMA APIs Description 133
Embedded Linux 136
11.1 Introduction 136
11.2 Linux 137
11.2.1 Linux History 137
11.2.2 Why Linux? 138
11.2.3 The difference between Linux and Windows 140
11.3 Embedded Linux 140

11.4 Embedded Linux Architecture 141
11.4.1 Toolchain 142
11.4.2 Bootloader 144
11.4.3 Kernel 149
11.4.4 Root Filesystem 150
11.4.5 Specific Application 151
11.5 Build Systems 152
11.5.1 Buildroot 152
11.5.2 Yocto / OpenEmbedded 152
11.5.3 OpenWRT / LEDE 153
11.5.4 AOSP / Soong 153
11.6 The Yocto Project 153
11.7 Summary 157
Embedded Linux Role in V2V Collision Avoidance 158
12.1 Introduction to Yocto Project 159
12.2 Setting up an Ubuntu Host 161
12.3 Build & run your first ever image 164
12.4 Building a basic image for RPI 167
12.5 Creating and adding a new layer to your image 171
12.6 Creating your first custom recipe 177
12.7 Customizing images by adding your recipes 186
12.8 Summary 189
References 190
Project Summary in Arabic / الملخص العربي 192

List of figures:

Introduction	
Figure 0.1	The V2V System.
Figure 0.2	EEBL System.
Figure 0.3	FCW System.
Figure 0.4	BSW System.
Figure 0.5	DNPW System.
Figure 0.6	IMA System.

Chapter 1	
Figure 1.1	Raspberry pi.
Figure 1.2	A summary comparison of commonly available RPI models.
Figure 1.3	Processor of Raspberry pi.
Figure 1.4	Ram memory of Raspberry pi.
Figure 1.5	Ethernet controller of Raspberry pi.
Figure 1.6	USB controller of Raspberry pi.
Figure 1.7	Power circuitry of Raspberry pi.
Figure 1.8	GPIO pins of Raspberry pi.

Chapter 2	
Figure 2.1	the Two-Way Communication
Figure 2.2	One "Master"-Multiple "Slave" *Communication*.
Figure 2.3	One "Slave"-Multiple "Master" *Communication*.
Figure 2.4	The Module ESP32.
Figure 2.5	Connect The ESP32 With Raspberry Pi.
Figure 2.6	UART Frame.
Figure 2.7	UART Connection.

Chapter 3	
Figure 3.1	LCD Module.
Figure 3.2	HC-SR04 Ultrasonic Module.
Figure 3.3	Ultrasonic Operation condition.
Figure 3.4	HC-SR04 module Timing Diagram.
Figure 3.5	DC-Motor Types.
Figure 3.6	DC-Motor H-Bridge.
Figure 3.7	H-Bridge Circuit.
Figure 3.8	PWM Duty Cycle.
Figure 3.9	HMC5883L Single Supply.
Figure 3.10	HMC5883L Module.
Figure 3.11	BUZZER Module.
Figure 3.12	Water Level Circuit using Buzzer.

Chapter 4	
Figure 4.1	ESP Boards.
Figure 4.2	One Way Communication.
Figure 4.3	One Initiator Multiple Responders.
Figure 4.4	One Responder, Multiple Initiators.
Figure 4.5	Two-Way Communication.
Figure 4.6	Two-Way Network.
Figure 4.7	MAC Address.
Figure 4.8	Callback Functions.
Figure 4.9	Send Process.
Figure 4.10	Receive Process.

Chapter 5	
Figure 5.1	Two UARTs directly communicate with each other.
Figure 5.2	UART with data bus.
Figure 5.3	UART packet.
Figure 5.4	Data bus to the transmitting UART.
Figure 5.5	UART data frame at the Tx side.
Figure 5.6	UART transmission.
Figure 5.7	The UART data frame at the Rx side.
Figure 5.8	Receiving UART to data bus.
Figure 5.9	Microcontroller data sheet.

Chapter 6	
Figure 6.1	Bluetooth.
Figure 6.2	Bluetooth master/slave topologies.
Figure 6.3	MAC NO.
Figure 6.4	Serial Port Profile.
Figure 6.5	HID interface, from RN-42-HID User's Guide.
Figure 6.6	Image from *A2DP specification* (v1.3).
Figure 6.7	Image from AVRCP specification (v1.5).
Figure 6.8	The architecture of Bluetooth host and controller in ESP-IDP.
Figure 6.9	Configuration of the HCI IO interface.
Figure 6.10	VHCI configuration.
Figure 6.11	UART configuration.

Chapter 7	
Figure 7.1	OSI Model And ESP-NOW Model.
Figure 7.1	ESP-NOW Functions.
Figure 7.1	System layout.

Chapter 8	
Figure 8.1	EEBL.
Figure 8.2	FCW.
Figure 8.3	EEBL Scenario.
Figure 8.4	FCW Scenario.
Figure 8.5	Flow Chart of Main Algorithm.
Figure 8.6	Pseudo Code of Main Algorithm.
Figure 8.7	Block Diagram of Main Algorithm.
Figure 8.8	Flow Chart of ID-Algorithm.
Figure 8.9	Block Diagram of ID- Algorithm.
Figure 8.10	Pseudo Code of ID- Algorithm.
Figure 8.11	EEBL&FCW Context Diagram.

Chapter 9	
Figure 9.1	BSW.
Figure 9.2	DNPW.
Figure 9.3	Flow chart of main algorithm.
Figure 9.4	Pseudo code of main algorithm.
Figure 9.5	Block diagram of main algorithm.
Figure 9.6	Flow chart of id algorithm.
Figure 9.7	Pseudo code of id algorithm.
Figure 9.8	BSW-DNPW layered architecture.

Chapter 10	
Figure 10.1	IMA Description.
Figure 10.2	Flow Chart of Main Algorithm.
Figure 10.3	Pseudo Code of Main Algorithm.
Figure 10.4	Block Diagram of Main Algorithm.
Figure 10.5	Flow Chart of ID Algorithm
Figure 10.6	Pseudo Code of ID Algorithm.
Figure 10.7	IMA Context Diagram.

Chapter 11	
Figure 11.1	GNU/Linux OS.
Figure 11.2	Embedded Linux Architecture.
Figure 11.3	Types of Toolchains.
Figure 11.4	Cross-compilation toolchain.
Figure 11.5	Booting Sequence.
Figure 11.6	GNU GRUB.
Figure 11.7	Raspberry pi boot sequence.
Figure 11.8	Kernel main jobs.
Figure 11.9	YOCTO project components.

List of tables

Chapter 3	
Table 3.1	Pins configuration.
Table 3.2	DDRAM Address Map.
Table 3.3	LCD Commands.
Table 3.4	HMC5883L Pin Configurations.

Chapter 5	
Table 5.1	UART Summary.
Table 5.2	Baud Rate Example Based on 26 MHz PCLK.
Table 5.3	Baud Rate Example Based on 16 MHz PCLK.
Table 5.4	UART Register Descriptions.

Chapter 11	
Table 11.1	Linux vs Windows.
Table 11.2	Architectures of different bootloaders.

Table of An Abbreviations

Abbreviation	Definition
A	
ADAS	Advanced Driver-assistance System
API	Application Programming Interface
ADC	Analog to Digital Converter
A2DP	Advanced Audio Distribution Profile
AVRCP	Audio/video Remote Control Profile
B	
BSW	Blind Spot Warning
BLE	Bluetooth Low Energy
C	
CPU	Central Processing Unit
D	
DNPW	Do Not Pass Warning
DAC	digital to analog converter
E	
EEBL	Emergency Electronic Brake Lights
ECU	Electronic Control Unit
EXT	Extended File System
F	
FCW	Forward Collision Warning
FHA	Federal Highway Administration

Abbreviation	Definition
G	
HFP	Hands-Free Profile
HLD	High Level Design
HID	Human Interface Device
G	
HID	Human Interface Device
HLD	High Level Design
I	
IEEE	Institute of Electrical and Electronics Engineers
IMA	Interaction Movement Assist
IOT	Internet of Things
I2C	Inter-Integrated Circuit
L	
LXDE	Lightweight X11 Desktop Environment
LED	Light Emitting Diode
LCD	liquid-crystal display
M	
MAC	Media Access Control
MC	Microcontroller
O	
OSI	Open Society Institute
OE-Core	Open Embedded-Core
Q	
PWM	Pulse-width modulation

Abbreviation	Definition
Q	
QA	Quality Assurance
R	
RPi	Raspberry Pi
RTOS	Real-Time Operating System
S	
SPI	Serial Peripheral Interface
SPP	Serial Port Profile
SDK	Software Development Kit
SOC	system on chip
T	
TSMC	Taiwan Semiconductor Manufacturing Company Limited
TCP/IP protocol	Transmission Control Protocol/Internet Protocol
U	
UART	Universal Asynchronous Receiver Transmitter
V	
V2V	Vehicle-To-Vehicle

Introduction

0

0.1V2V Communication

0.1.1 What is V2V?

- Vehicle-to-vehicle (V2V) communication's ability to wirelessly exchange information about the speed and position of surrounding vehicles shows great promise in helping to avoid crashes, ease traffic congestion, and improve the environment [1].
- But the greatest benefits can only be achieved when all vehicles can communicate with each other [1].
- Using vehicle-to-vehicle (V2V) communication, a vehicle can detect the position and movement of other vehicles up to a quarter of a kilometer away [1].

- In a real world where vehicles are equipped with a simple antenna, a computer chip and GPS (Global Positioning System) Technology, your car will know where the other vehicles are, additionally other vehicles will Know where you are too whether it is in blind spots, stopped ahead on the highway but hidden from view, around a blind corner or blocked by other vehicles [1].
- The vehicles can anticipate and react to changing driving situations and then instantly warn the drivers with emergency warning messages [1].
- If the driver doesn't respond to the alerts message, the vehicle can bring itself to a safe stop, avoiding a collision [1].

0.1.2 V2V Market

- V2V communication ‘s market is growing every year due to the enhancement of technology use in vehicles [1].

- A lot of investment is done in this field nowadays, Middle Eastern countries are considered a great potential for this technology due to the increase in population as well as the focus of many automobile companies on regions such as the Middle East and Africa [1].

- The development of this technology by automotive manufacturers, chip manufacturers as well as technology and solution providers are accelerating [1].

0.1.3 Competition Landscape

- There are many companies interested in this technology such as BMW, Audi, Daimler, Volvo, and Ford App link. Among the solution providers Trans Systems, Qualcomm Technologies Inc., Cisco Systems Inc., Delphi Automotive PLC, Auto talks Ltd., Denso, Arado Systems, Kapsch Group and Savarin Inc., are included in the vehicle-to-vehicle communication market [1].

0.1.4 Standardized V2V Protocol

- Since V2X requires devices and vehicles of different manufacturers to communicate with each other, there must be a standard that all companies and manufacturers will follow. That's why IEEE developed the 802.11p standard which explains the physical and mac layers of vehicular transceivers [1].

- That way, any other European or American standards developed, will have to be based on the lower-level IEEE 802.11p standard, to ensure the compatibility of different devices communicating with each other [1].

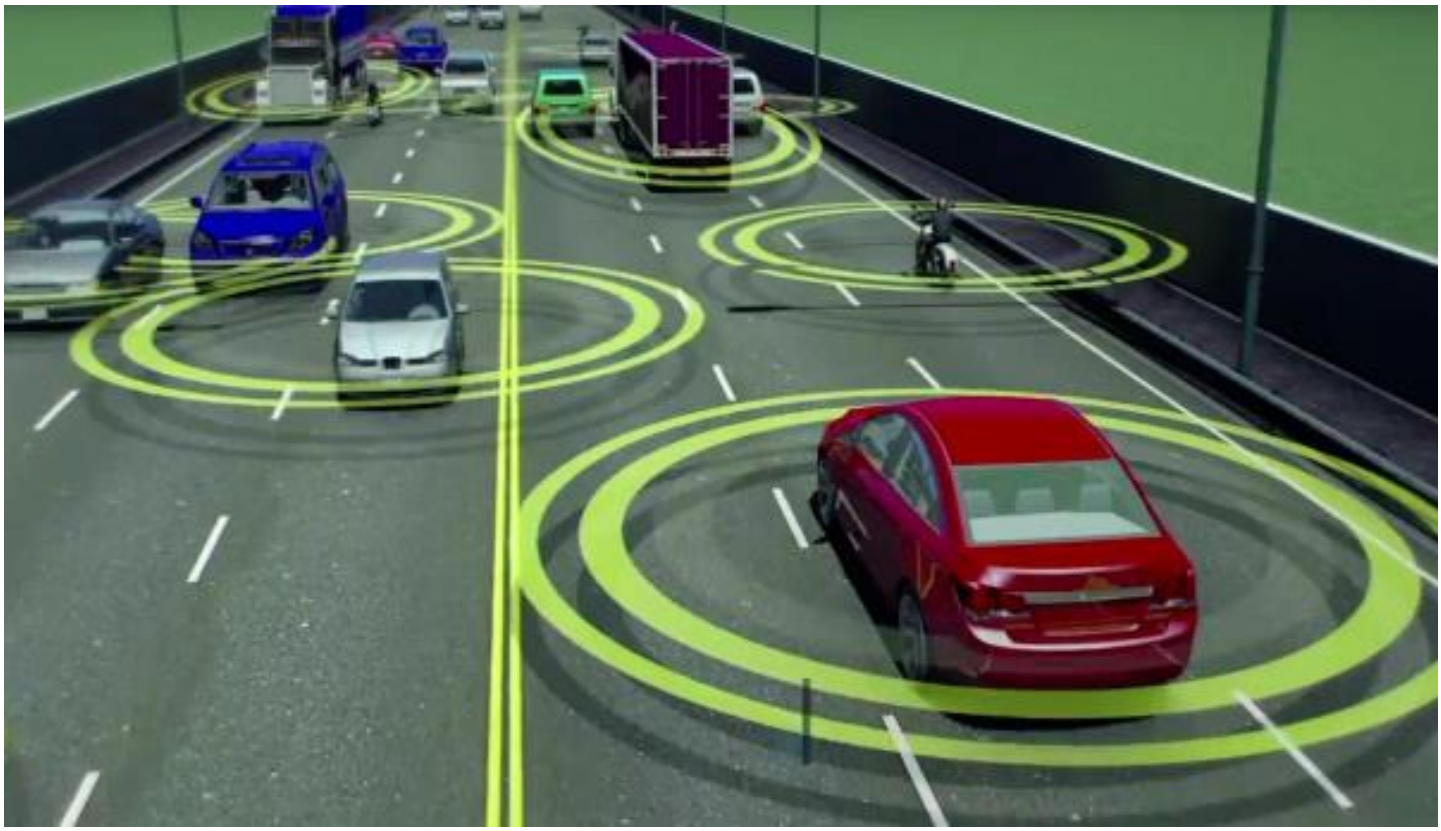


Figure 0.1 the V2V System

0.2 Project Objectives

- The objective of vehicle-to-vehicle Communication System project is to avoid roadway collisions and to drivers prior to the collision.

- Accidents are taking thousands of lives every day so with intent to reduce the adverse effects we implemented this project based on vehicle-to-vehicle communications systems, with the rapid advancement of wireless communication systems had paved path for the vehicle-to-vehicle wireless communication-based warning systems.

We also wanted our project to fulfill the user's needs and expectations and satisfy:

- Flexibility and compatibility that it can implemented on any car type or model.

- Cheap and easy user interface that if we want to update the system anytime it will be so easy because it is programmable, so it doesn't need any hardware exchange.
- Guaranteeing safety to people on the road.

- Regulation of cars motion so that no car can detect the other cars motion information in its pre-specified range.

0.3 Motivation

Our Thought During Idea Searching Phase We thought about a project that meets our jobs requirements and meets our team's intended learning objective through the graduation project, and we chose an idea which assembles different skills in our team.

In addition, this project will help in solving economic problems and tragic problems caused by road accidents, it will provide safety to the drivers, and it will improve traffic management.

0.4 Project Description

- The great ability of the machine on Focusing and not be distracted made us use this as a system, this system will provide a great help to reduce road accidents and congestion by making communication between vehicles to collect surrounding vehicles' motion information this information is one of the keys for accident prevention by helping the driver to make the right decision in the right time [2].

- This is what Vehicle to Vehicle communication systems do, V2V communications are going to create new services by transmitting packets from vehicle to vehicle without the use of any deployed infrastructures [2].

- These services will enable the vehicle to transmit necessary data such as the current location, the motion's direction, and the speed directly to other vehicles, these vehicles would form a network, and pass information about road conditions, accidents, and congestion [2].

- A driver could be made aware of the emergency braking of a preceding vehicle or the presence of an obstacle in the roadway, it can also help vehicles negotiate critical points like blind crossings at the intersections or entries to highways, then vehicles can react to change driving situations and then instantly warn the drivers with emergency warning messages [2].

- Using vehicle-to-vehicle (V2V) communication, a vehicle can detect the position and movement of other vehicles up to a quarter of a kilometer away [2].

- In area world where vehicles are equipped with a computer chip, GPS (Global Positioning System) Technology and some other sensors to detect vehicle information, your car will know where the other vehicles are, additionally other vehicles will Know where you are too whether it is in blind spots, stopped ahead on the highway but hidden from view, around a blind corner or blocked by other vehicles [2].

- The vehicles can anticipate and react to changing driving situations and then instantly warn the drivers with emergency warning messages [2].

- So, our project elaborates Implementation of a real time vehicle to vehicle communication system over a local WIFI network between cars where we broadcast car data and state to avoid potential collisions**. Specifically handling 5 main applications:**

1-EMERGENCY ELECTRONIC BRAKE LIGHTS (EEBL):

- Every single day we always face the following situation, when there is a hard-braking vehicle in the pass ahead, or any object just came out of nowhere in front of another vehicle, the driver of this vehicle may not be able to get a proper chance to stop car and avoid collisions that may occur, Here comes the use of our system "Emergency Electronic Brake Light" [2].

- When such a situation occurs, The Rear vehicle will automatically send a warning through an actuator to the driver [2].

Figure 0.2 EEBL System

2-FORWARD COLLISION WARNING(FCW):

- Rear-end crashes are one of the most common types of collisions, they are often the result of drivers following too closely or drivers that are distracted or speeding [2].

- In fact, 40% of rear-end collisions have no brake application whatsoever, reflecting driver distraction, and The NTSB has found that over 80% of rear-end crashes could be prevented with forward collision avoidance systems [2].

- The driver whose car hit the car ahead is typically considered to have been at-fault [2].

- Here comes the use of our system "Forward Collision Warning [2].

- When such a situation occurs, The Rear vehicle will automatically send a warning through an actuator to the driver [2].

- And both cars still getting closer, then for a certain distance the ECU will automatically brake the care without any control from the driver side [2].

Figure 0.3 FCW System

3-BLIND SPOT WARNING(BSW):

When there is a car that may not be visible to the car driver, because of v2v Communication a warning message is issued to make you aware of the presence of This vehicle , should you attempt to lane change this warning message will tell you That it's not safe to lane change [2].

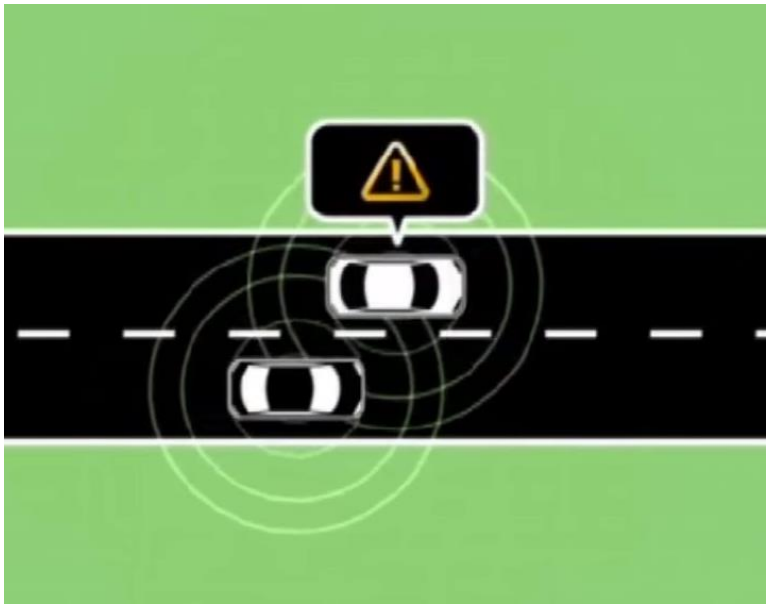


Figure 0.4 BSW System

4_ DO NOT PASS WARNING(DNPW):

The Do Not Pass Warning (DNPW) application is intended to warn the driver of the vehicle during a passing maneuver attempt when a slower moving vehicle, ahead and in the same lane, cannot be safely passed using a passing zone which is occupied by vehicles in the opposite direction of travel [2].

The system sends a warning message to the driver in order to prevent the crash. It may also stop the car if the distance was very small [2].

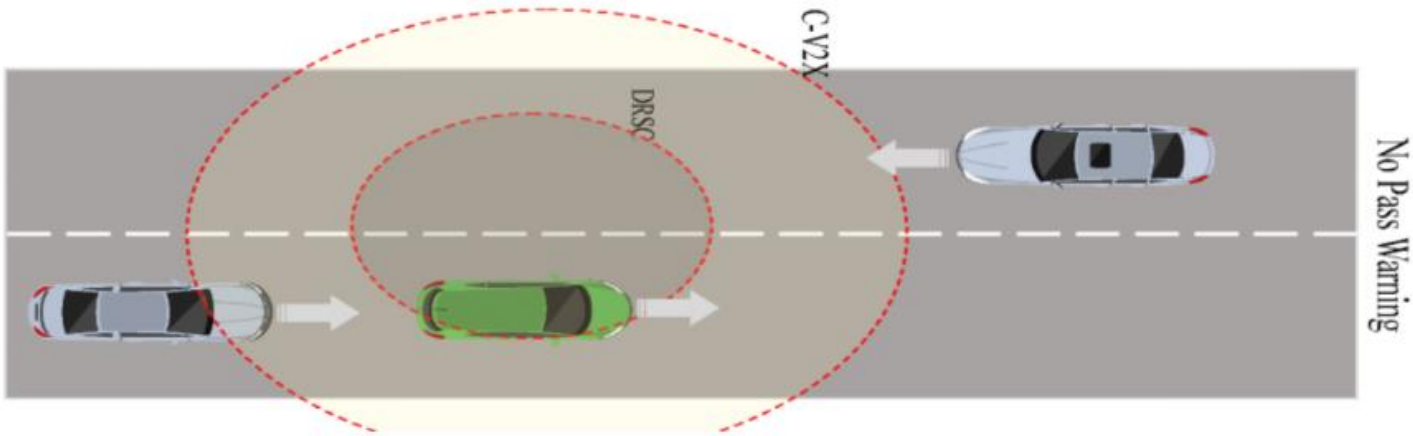


Figure 0.5 DNPW System

5-INTERSACTION MOVEMENT ASSIST(IMA):

The Intersection Movement Assist (IMA) application warns the driver of a vehicle when it is not safe to enter an intersection due to high collision probability with other vehicles at stop sign controlled and uncontrolled intersections [2].

This application can provide collision warning information to the vehicle operational systems which may perform actions to reduce the likelihood of crashes at the intersections [2].

Figure 0.6 IMA System

0.5 Research Methodology, Implementation and Validation

The methods that will use to finish this project are enlisted here.

1-Learn and study the Linux operating system (Linux administration):

- Booting Process
- System Calls
- Users
- File System
- Commands
- Mount Devices
- Process Management

And making a good knowledge in it, then use this knowledge to deal with Linux operating system professionally in our project.

2-Learn and study the basic concepts of Bash scripting and use it to develop shell scripts needed.

3-Learn and study the basic concepts of device drivers, then use it to implement the device drivers needed like (LED, Ultrasonic , LED… device drivers) so we can make the interfacing between the sensors and the OS.

4-Learn the study of the Raspberry Pi as a embedded Linux board and how to burn our image on it.

5-Learn and study Customization and how to create our distribution(image) from Linux operating system to be suitable with our project needed.

6-Connect all parts of the systems and test the communication between them.

7-Making hardware prototype and testing the System function.

0.6 Development Plan

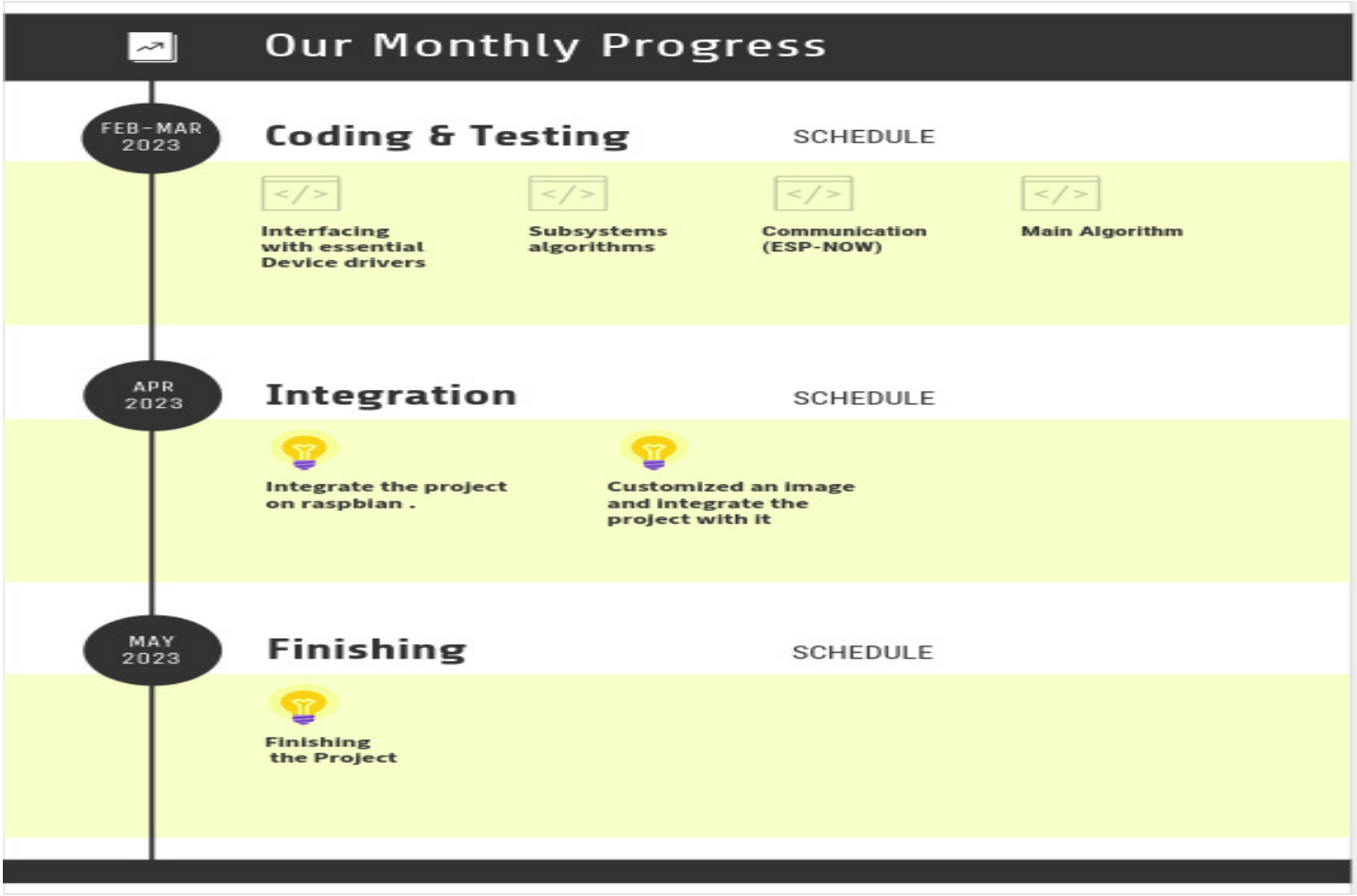
At the begging of any project, we need to consider a development plan contains well defined phases and deadlines to assure utilization of time during the project as well as Global design document which includes deep technical details about the whole project.

0.6.1 Project Timeline

This part includes our project deliverables, durations, deadlines, milestones of our project.

Project timeline was developed on JIRA, a tool used by professional project managers to develop timelines and other matters for moderate size projects. Full project management plan attached in references.

In this project we follow agile methodology in our project development process to assure the best results in the shortest time.



0.6.2 Global design Document (GDD)

These documents are:

1. The High-Level Design Document (HLD) :
   - Goals and Objectives
   - Terms and Abbreviations
   - System Description
   - Architectural Design
   - Software Components
   - Sequence Diagrams
   - Software Integration Constraints

2. Components Design Document (CDD) For each Component:
   - Software Context
   - External Interfaces
   - Design Constraints
   - Design Choices and Alternatives
   - Components Design
   - Components Constraints

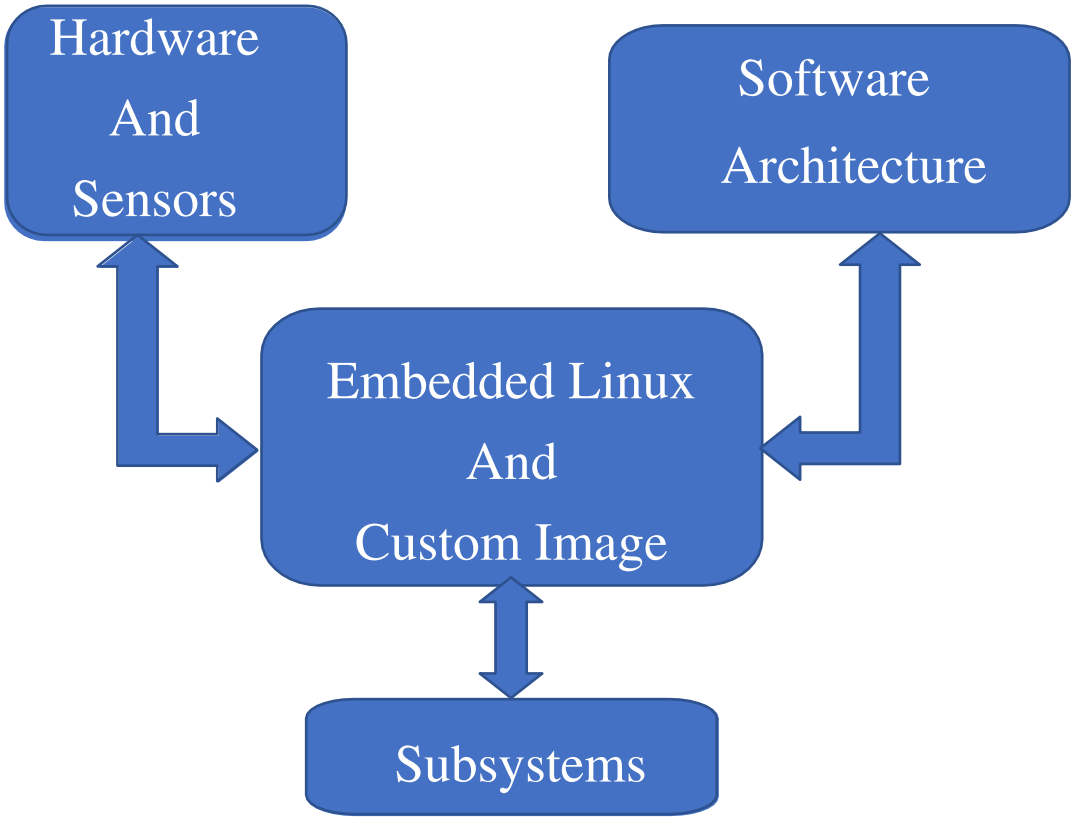
For our graduation project identified by project team members.

0.7 Project Architecture

Our Project contains 4 modules:

- Model A: Hardware and Sensors
- Module B: Software Architecture
- Module C: Subsystems
- Module D: Embedded Linux and Custom Image

We'll talk about them individually.



Module A: Hardware and Sensors

We'll explain the hardware used, the main ECU is the Raspberry Pi and components of it, the communication ECU is the ESP32 and the components, the Sensors which connected to the overall system.

This module consists of three chapters, listed as:

Chapter 1: The Main ECU (The Raspberry Pi).

Chapter 2: The Communication ECU (ESP32).

Chapter 3: The Sensors and Actuators.

Module B: Software Architecture

We'll explain the software architecture used, which is the communication between the ESP32 boards and the way we communicate between the raspberry pi and ESP32 to get the messages correctly.

This module consists of four chapters, listed as:

Chapter 4: Peer to Peer ESP32 Communication (ESP NOW).

Chapter 5: Universal Asynchronous Receiver Transmitter (UART).

Chapter 6: Bluetooth.

Chapter 7: Communication between ESP32 and Raspberry Pi.

Module C: Subsystems

We'll explain the Subsystems used to solve the problem of collision and crash, these are 5 Applications (EEBL , FCW , BSW , DNPW and IMA) and explain the function of each subsystem and the role of it in the system overall.

This module consists of three chapters, listed as:

Chapter 8: EEBL And FCW.

Chapter 9: BSW And DNPW.

Chapter 10: IMA.

Module D: Embedded Linux And Custom Image

We'll explain what the meaning of embedded Linux and how we use it in our project and explain the customization process and how to create the custom image.

This module consists of two chapters, listed as:

Chapter 11: Embedded Linux

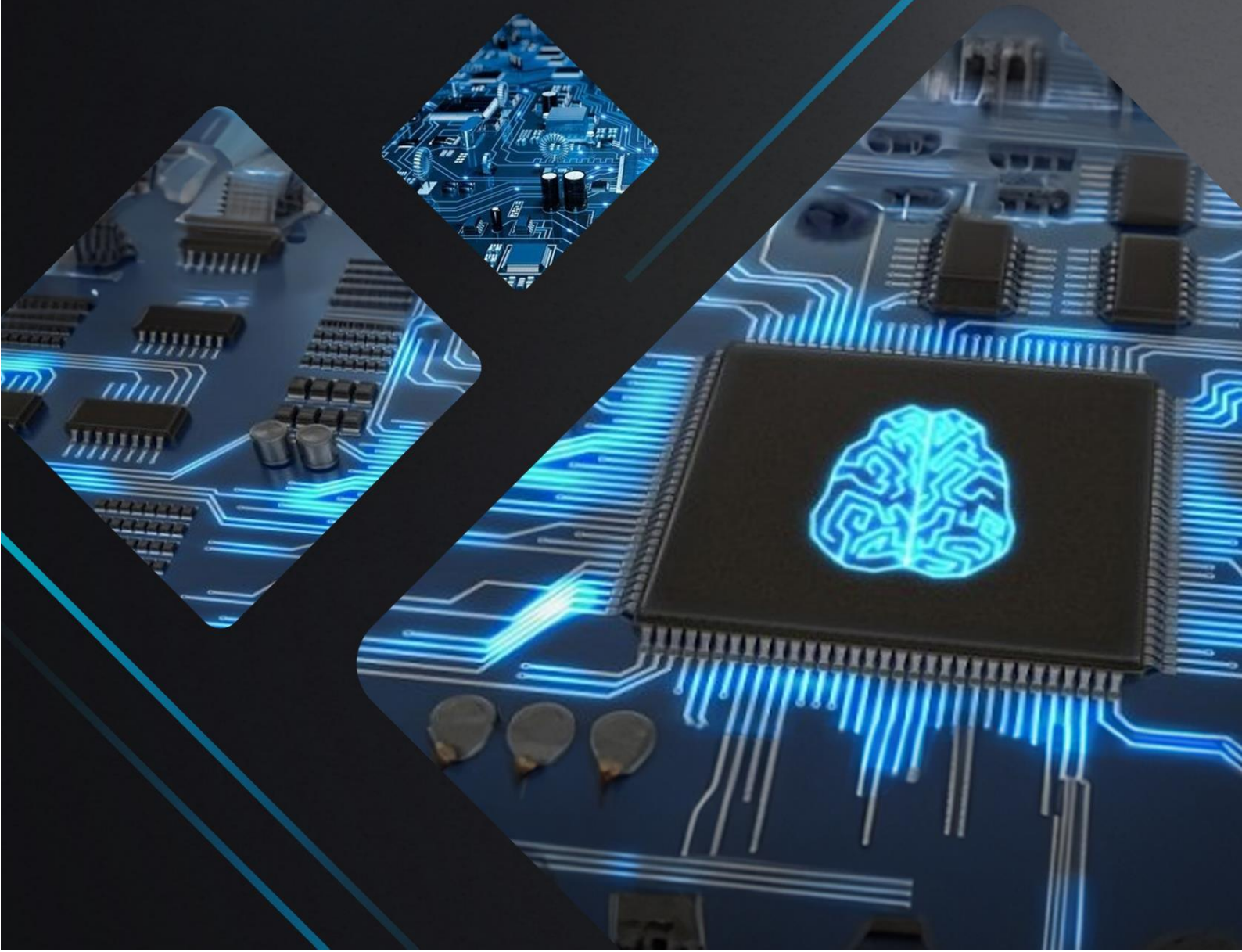
Chapter 12: Embedded Linux Role in V2V Collision Avoidance

I A

HARDWARE COMPONENTS

CONTENT :

Chapter1: Main ECU Raspberry pi
Chapter2: Communication ECU(ESP32)
Chapter3: Sensors & Actuatros



Main ECU (Raspberry pi)

1

In this chapter, we'll talk about the main ECU in our project which is Raspberry pi. Also, we'll discuss why Raspberry pi, its Versions, Hardware Component, Interfacing with GPIO pins and Raspberry pi Operating System.

1.1 Introduction

Raspberry Pi is a low-cost, credit card-sized computer that was first developed in 2012 by the Raspberry Pi Foundation in the UK. The Raspberry Pi was designed to be a tool to promote the teaching of basic computer science in schools and developing countries [3].

The original Raspberry Pi was equipped with a 700 MHz ARM11 processor, 256 MB of RAM, and had a single USB port, however, today's models are much more powerful, with quad-core processors, up to 8GB of RAM, and multiple USB and HDMI ports [3].

The Raspberry Pi can run a variety of operating systems, including Linux distributions, Windows 10, and even Android, it can be used for a wide range of projects, from simple tasks such as browsing the web and playing games, to more complex projects such as controlling robots, home automation, and even building a supercomputer [3].

Overall, the Raspberry Pi has become an incredibly versatile and useful tool for anyone interested in learning about computing and electronics. There're multiple types of mobile robotics such as wheeled, tracked, legged, flying, and underwater robots [3].

We used Raspberry pi 4 in our project <u>Raspberry pi 4</u> :

- The speed and performance of the new Raspberry Pi 4 is a step up from earlier models. For the first time, Raspberry Pi 4 is a complete desktop experience. Whether you're editing documents, browsing the web with a bunch of tabs open, juggling spreadsheets or drafting a presentation, we'll find the experience smooth and very recognizable, but on a smaller, more energy- efficient and much more cost-effective machine [3].

  Raspberry Pi 4 features can list as:

  - Silent, energy-efficient:
    The fan less, energy-efficient Raspberry Pi runs silently and uses far lesspower than other computers [3].
  - Fast networking: Raspberry Pi 4 comes with Gigabit Ethernet, along withonboard wireless networking and Bluetooth [3].
  - USB 3: Raspberry Pi 4 has upgraded USB capacity: along with two USB2 ports we'll find two USB 3 ports, which can transfer data up to ten times faster [3].
  - Choice of RAM: There are different variants of the Raspberry Pi 4 available, depending on how much RAM you need — 1GB, 2GB, 4GB, or 8GB [3].

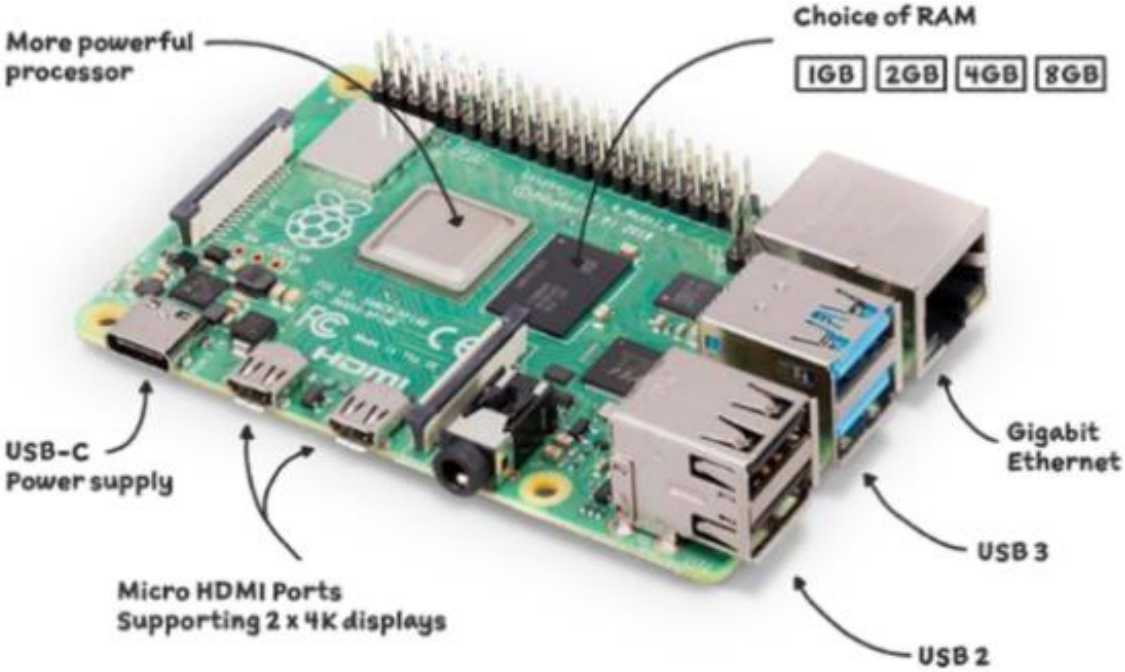


Figure 1.1 Raspberry pi

1.2 Why Raspberry pi

Raspberry Pi has many advantages that make it a popular choice for a wide range of projects and applications. Here are some reasons why Raspberry Pi is a great choice:

- Low cost: Raspberry Pi is a low-cost computer, making it an affordable option for beginners or for use in projects where cost is a concern [3].
- Small Size: The Raspberry Pi is small and lightweight, so it's easy to carry around and fit into tight spaces [3].
- Low Power Consumption: Raspberry Pi uses very little power, making it an energy-efficient option [3].
- Flexibility: The Raspberry Pi can run a variety of operating systems and applications, making it a versatile tool for a wide range of projects [3].
- GPIO Pins: The Raspberry Pi's GPIO pins make it easy to connect to a wide range of sensors and devices, allowing you to build custom electronics projects [3].
- Community Support: Raspberry Pi has a large and active community of users and developers who share knowledge and resources, making it easy to get help and find solutions to problems [3].
- Educational Value: Raspberry Pi was designed as a tool to promote the teaching of basic computer science in schools and developing countries, so it's a great way to learn about computing and electronics [3].

Overall, Raspberry Pi is a cost-effective, flexible, and versatile tool that can be used for a wide range of projects and applications, making it a popular choice for hobbyists, students, and professionals alike [3].

1.3 Raspberry pi Versions

Here are the different versions of Raspberry Pi that have been released so far, each version of the Raspberry Pi has its own unique features and capabilities, and they can be used for a wide range of applications, including education, hobby projects, and industrial automation:

- Raspberry Pi Model A: Released in 2012, it was the first Raspberry Pi board and featured a 700MHz ARM11 processor, 256MB of RAM, one USB port, and no Ethernet port [3].

- Raspberry Pi Model B: Released in 2012, it was a higher-end version of the Model A and featured a 700MHz ARM11 processor, 512MB of RAM, two USB ports, and an Ethernet port [3].
- Raspberry Pi Model A+: Released in 2014, it was a smaller and cheaper version of the Model A, with 256MB of RAM, one USB port, and no Ethernet port [3].
- Raspberry Pi Model B+: Released in 2014, it featured a 900MHz quad-core ARM Cortex-A7 processor, 1GB of RAM, four USB ports, and an Ethernet port [3].
- Raspberry Pi 2 Model B: Released in 2015, it featured a 900MHz quad-core ARM Cortex-A7 processor, 1GB of RAM, four USB ports, and an Ethernet port [3].
- Raspberry Pi Zero: Released in 2015, it was a smaller and cheaper version of the Model A, with a single-core 1GHz ARM11 processor, 512MB of RAM, and a micro-USB port [3].
- Raspberry Pi 3 Model B: Released in 2016, it featured a 1.2GHz 64-bit quad-core ARM Cortex-A53 processor, 1GB of RAM, four USB ports, and an Ethernet port [3].
- Raspberry Pi Zero W: Released in 2017, it was the same as the Zero but with added wireless connectivity, including Wi-Fi and Bluetooth [3].
- Raspberry Pi 3 Model B+: Released in 2018, it featured a 1.4GHz 64-bit quad-core ARM Cortex-A53 processor, 1GB of RAM, four USB ports, and an Ethernet port [3].
- Raspberry Pi 4 Model B: Released in 2019, it featured a 1.5GHz 64-bit quad-core ARM Cortex-A72 processor, up to 8GB of RAM, two USB 2.0 ports, two USB 3.0 ports, and two micro-HDMI ports [3].
- Raspberry Pi Compute Module 3: Released in 2017, it was designed for industrial applications and included a 1.2GHz quad-core ARM Cortex-A53 processor, 1GB of RAM, and 4GB of eMMC flash storage [3].
- Raspberry Pi 400: Released in 2020, it featured a Raspberry Pi 4 board built into a compact keyboard, with 4GB of RAM, two USB 3.0 ports, and one USB 2.0 port [3].
- Raspberry Pi Compute Module 4: Released in 2020, it was the latest version of the Compute Module and included a choice of processors, up to 8GB of RAM, and various storage options [3].

Figure 1.2 provides a summary feature comparison of the different RPi models that are presently available. Here is a quick summary of this table:

- If you need an RPi for general-purpose computing, consider the RPi 3.

The 1 GB of memory and 1.2 GHz quad-core processor provide the best performance out of all the boards [3].

- For applications that interface electronics circuits to the Internet on a wired network, consider the RPi 3, RPi 2, or RPi B+, with cost being the deciding factor [3].
- If you need a small-footprint device with wireless connectivity, consider the RPi Zero. The RPi A+ could be used to develop the initial prototype [3].
- If you want to design your own PCB that uses the RPi (or multiple RPi boards), investigate the Compute module [3].

Model	RPi 3	RPi 2	RPi B+	RPi A+	RPi Zero	RPi B	Compute
Characteristics	performance/Wi-Fi Bluetooth/Ethernet	performance/Ethernet	Ethernet	price	price/size	original	integration/eMMC
Price	$35	$35	$25	$20	$5+	$25	$40 ($30 volume)
Processor*	BCM2837 quad core Linux ARMv7	BCM2836 quad core Linux ARMv7	BCM2835 Linux ARMv6	BCM2835 Linux ARMv6	BCM2835 Linux ARMv6	BCM2835 Linux ARMv6	BCM2835 Linux ARMv6
Speed	1.2 GHz	900 MHz	700 MHz	700 MHz	1 GHz	700 MHz	700 MHz
Memory	1 GB	1 GB	512 MB	256 MB	512 MB	512 MB	512 MB
Typical power	2.5 W (up to 6.5 W)	2.5 W (up to 4.1 W)	1 W (up to 1.5 W)	1 W (up to 1.5 W)	1 W (up to 1.5 W)	1 W (up to 1.5 W)	1 W (up to 1.5 W)
USB Ports	4	4	4	1	1 OTG	2	via header
Ethernet	10/100 Mbps, Wi-Fi, and Bluetooth	10/100 Mbps	10/100 Mbps	none	none	10/100 Mbps	none
Storage	micro-SD	micro-SD	micro-SD	micro-SD	micro-SD	SD	4 GB eMMC
Video	HDMI composite	HDMI composite	HDMI composite	HDMI composite	mini-HDMI composite	HDMI RCA video	HDMI via edge TV DAC via edge
Audio	HDMI digital audio and analog stereo via a 3.5 mm jack (where available)						via edge connector
GPU	Dual Core VideoCore IV Multimedia Co-Processor at 250 MHz (24 GFLOPS)						
Camera (CSI)	yes	yes	yes	yes	no	yes	CSI x 2 via edge
Display (DSI)	yes	yes	yes	yes	no	yes	DSI x 2 via edge
GPIO header	40 pins	40 pins	40 pins	40 pins	40 pins	26 pins	48 pins via edge
Usage	General-purpose computing and networking. High-performance interfacing. Video streaming	General-purpose computing. High-performance interfacing. Video streaming	General-purpose computing. Internet-connected host. Video streaming	Low-cost general-purpose computing. Standalone electronics interfacing applications	Low-cost small-profile standalone electronics interfacing projects	General-purpose legacy applications. Internet-connected host	Suitable for plugging into user-created PCBs using a DDR2 SODIMM connector. Open-source breakout board available

Figure 1.2 A summary comparison of commonly available RPi models

1.4 Raspberry pi Hardware

In this Part, we'll talk about the **Raspberry pi 4** Specifications Pin Diagram and Description which we are used it in our project. Also, we'll discuss the Layout and Interfacing of GPIO pins and Hardware Components of Raspberry pi.

1.4.1 Specifications Pin Diagram and Description

Here are the specifications of the Raspberry Pi 4 Model B, the latest version of the Raspberry Pi as of my knowledge cut-off in September 2021:

- It has a 64bit quad-core processor having cortex-A72 (ARM v8) clocked @1.5GHz [3].
- The new Pi board includes Broadcom BCM2711 VC6 GPU able to handle two 4kp30 displays also it can handle H.265 decoding at 4kp60. It has two micro-HDMI ports [3].
- Now the new Pi board comes in 2GB, 4GB, and 8GB LPDDR-4 RAM options [3].
- It has a dual-band 2.4/5.0 GHz Wi-Fi, Bluetooth 5.0, Gigabit Ethernet Port, 2 USB 3.0, and 2 USB 2.0 ports [3].
- USB type C power input port. Also, it has POE capability via separate POE HAT (add-on) [3].
- It has a standard 40 pin GPIO header (having backward compatibility)
- The new Raspberry Pi is even more fast, powerful, and versatile for a huge variety of projects involving robotics, automation, image processing, AI, and many more [3].

1.4.2 Hardware Components of Raspberry pi

Raspberry Pi is a credit card-sized computer that can be used for a variety of projects, including robotics, home automation, and media centers. It consists of several hardware components, including:

1. Processor Of Raspberry: Pi has a more powerful quad-core Cortex-A72 64-bit CPU running at 1.5GHz and Broadcom Video Core VI GPU running at 0.5GHz. It can handle 4k videos, H.265
   encoding 2 HDMI outputs. It has a fully functional Gigabit Ethernet interface and USB 3.0 port [3].

Figure 1.3 Processor of Raspberry pi

2. Ram Memory:
   The new Raspberry Pi 4 comes with an option of 2GB, 4GB, and 8GB of RAM options. It has LPDDR4 RAM. Also, you get a heat sink for the RAM chip as it gets hot while operating [3].

Figure 1.4 Ram Memory of Raspberry pi

3. Ethernet Controller
   This Pi board has a Broadcom BCM54213 Gigabit ethernet controller. It falls to a slower speed if the network speed is slow [3].
   - Software-driven router
   - Media server
   - A bridge between Wi-Fi and Ethernet networks

Figure 1.5 Ethernet Controller of Raspberry pi

4. USB Controller

   VL805 chip is the USB port controller for Raspberry Pi 4. The Pi has 2 x USB 3.0 and 2 x USB 2.0 ports [3].

   VL805 chip is a USB 3.0 Host controller. it allows the PCI Express platform to interface with USB Super-Speed (5 Gbps), High-Speed (480 Mbps), Full-Speed (12 Mbps), and Low-Speed (1.5 Mbps) devices [3].

   The root hub is consisting of four downstream facing ports, it enables simultaneous operation of multiple peripheral devices [3].

Figure 1.6 USB Controller of Raspberry pi

5. Power Circuitry

   The Pi4 uses the MXL7704 it was developed specifically for Raspberry [3].

   This chip reduced the complexity of the Pi board and delivers more power to all the peripheral to run at more power [3].

   Due to this chip cost of the board has also been reduced as the number of components has been reduced [3].

Figure 1.7 Power Circuitry of Raspberry pi

1.4.3 Layout of The GPIO header

The GPIO header allows users to connect a wide range of devices and peripherals to the Raspberry Pi, making it highly versatile and flexible for a wide range of projects [4].

And here is a pin diagram of the Raspberry Pi 4 Model B, showing the layout of the GPIO header there are 40 GPIO Pins of Raspberry pi 4:

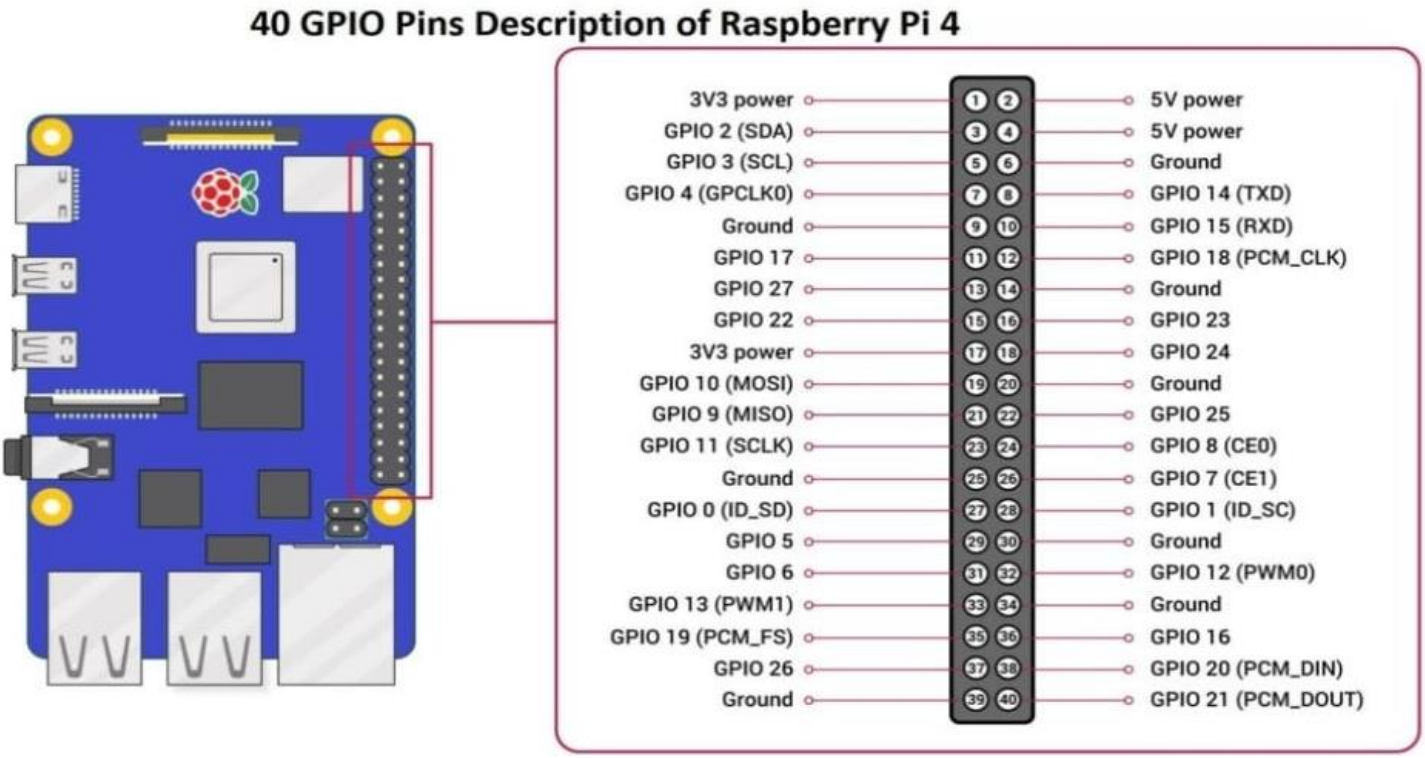


Figure 1.8 GPIO Pins of Raspberry pi

1.5 Raspberry pi OS

Raspberry Pi OS (formerly known as Raspbian) is the official operating system for the Raspberry Pi [4].

It is a free and open-source operating system based on the Debian Linux distribution [4].

Raspberry Pi OS is specifically designed for the Raspberry Pi hardware and comes pre-installed with a variety of software [4].

Here are some of the features and components of Raspberry Pi OS:

1. Desktop Environment: Raspberry Pi OS comes with a desktop environment based on the LXDE desktop environment, which is lightweight and optimized for the Raspberry Pi's hardware [4].

2. Software: Raspberry Pi OS comes with a variety of pre-installed software, including the Chromium web browser, the Python programming language, and the Thonny Python IDE [4].

3. Configuration Tools: Raspberry Pi OS includes a variety of tools for configuring and customizing the operating system, including the Raspberry Pi Configuration tool and the raspi-config tool [4].

4. Command-Line Interface: Raspberry Pi OS includes a powerful command-line interface, which can be used to perform system administration tasks and run scripts.

5. Package Manager: Raspberry Pi OS includes the apt package manager, which can be used to install new software packages and keep the operating system up to date [4].

6. GPIO Support: Raspberry Pi OS includes support for the Raspberry Pi's GPIO pins, which can be used to interface with external hardware and sensors [4].

Overall, Raspberry Pi OS is a powerful and flexible operating system that is optimized for the Raspberry Pi hardware. It is a great choice for a variety of projects, from building a media center to creating a home automation system [4].

Communication ECU (ESP32)

2

In this chapter, we'll talk about how all vehicles can communicate with each other.

By using ESP32 that has two types of interfaces connections:

- Interface between ESP32 and itself in the other driving car.
- Interface between ESP32 and UART in same car.

2.1 ESP32

ESP32 is a single 2.4 GHz Wi-Fi and Bluetooth combo chip designed with TSMC low-power 40 nm technology. It is designed to achieve the best power and RF performance, robustness, versatility, and reliability in a wide variety of applications and different power profiles. ESP32 is a highly integrated solution for Wi-Fi + Bluetooth applications in the IOT industry with around 20 external components [5].

The ESP32 can be used in communication between cars through various wireless communication protocols including:

1. Wi-Fi: Both the ESP32 and Raspberry Pi have built-in Wi-Fi capabilities, which can be used to establish a wireless connection between the two devices. This can be done using the standard TCP/IP protocol [5].

2. Bluetooth: The ESP32 also has built-in Bluetooth capabilities, which can be used to establish a wireless connection with the Raspberry Pi. This can be done using the Bluetooth Low Energy (BLE) protocol [5].

3. MQTT: MQTT is a lightweight messaging protocol that is commonly used for IOT applications. It can be used to establish a wireless connection between an ESP32 and Raspberry Pi, allowing them to exchange data in real-time [5].

4. ESP-NOW: This protocol allows for fast and efficient communication between ESP32 devices without the need for a Wi-Fi network or internet connection. It is ideal for applications that require low-latency and low-power consumption, such as home automation, sensor networks, and remote-control systems. The ESP-NOW protocol is easy to use and can be implemented using development environments, Thus we will use ESP-NOW wireless communication protocol [5].

2.1.a ESP-NOW

- ESP-NOW is a connectionless communication protocol developed by Espresso that features short packet transmission [6].

- This protocol enables multiple devices to talk to each other without using Wi-Fi, this is a fast communication protocol that can be used to exchange small messages (up to 250 bytes) between ESP32 boards. ESP-NOW is very versatile, and you can have one-way or two-way communication in different arrangements [6].

We have other tutorials for ESP-NOW with the ESP32:

. ESP-NOW Two-Way Communication Between ESP32 Boards

. ESP-NOW with ESP32: Send Data to Multiple Boards (one-to-many)

. ESP-NOW with ESP32: Receive Data from Multiple Boards (many-to-one)

- ESP-NOW Two-Way Communication

With ESP-NOW, each board can be a sender and a receiver at the same time. So, you can establish two-way communication between boards [6].

Figure 2.1 the Two-Way Communication

- A “Master” ESP32 sending data to multiple ESP32 “Slaves.”

ESP32 board sending the same or different commands to different ESP32 boards. This configuration is ideal to build something like a remote control. You can have several ESP32 boards around the One house that are controlled by one main ESP32 board[6].

Figure 2.2 One "Master"-Multiple "Slave" Communication

- One ESP32 “Slave” receiving data from multiple “Masters”.

This configuration is ideal if you want to collect data from several sensors nodes into one ESP32 board. This can be configured as a web server to display data from all the other boards, for example [6].

Figure 2.3 One "Slave"-Multiple "Masters" Communication

2.2 Hardware ESP32

• The module ESP32 uses one single pin as the power supply pin. Users can connect the module to a 3.3 V power supply. The 3.3 V power supply works both for the analog circuit and the digital circuit [5].

• The EN pin is used for enabling the chip. Set the EN pin high for normal working mode [5].

• Lead the GND, RXD, TXD pins out and connect them to a USB-to-UART tool for firmware download, log-printing and communication, by default, the initial firmware has already been downloaded in the flash [5].

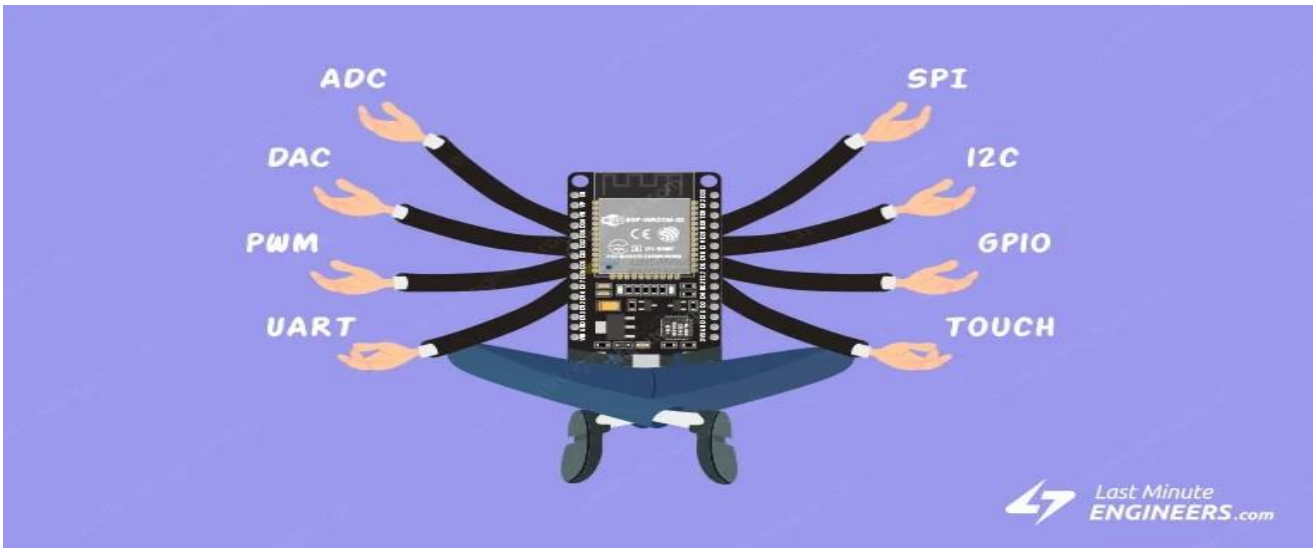
VIN GND D13 D12 D14 D27 D26 D25 D33 D32 D35 D34 VN VP EN
EN
BOOT
AMS1117
3.3 DN628
ESP-WROOM-32
3V3 GND D15 D2 D4 RX2 TX2 D5 D18 D19 D21 RX0 TX0 D22 D23

Figure 2.4 The Module ESP32

2.2.a ESP32 Peripherals and I/O

Although the ESP32 has 48 GPIO pins in total, only 25 of them are broken out to the pin headers on both sides of the development board [5]. These pins can be assigned a variety of peripheral duties, including:

15 ADC channels	15 channels of 12-bit SAR ADC with selectable ranges of 0-1V, 0-1.4V, 0-2V, or 0-4V
2 UART interfaces	2 UART interfaces with flow control and IrDA support
25 PWM outputs:	25 PWM pins to control things like motor speed or LED brightness
2 DAC channels:	Two 8-bit DACs to generate true analog voltages
SPI, I2C and I2S:	Three SPI and one I2C interfaces for connecting various sensors and peripherals, as well as two I2S interfaces for adding sound to your project
9 Touch Pads	9 GPIOs with capacitive touch sensing



2.3 How to Connect the ESP32 With Raspberry Pi

To connect the ESP32 with Raspberry Pi in hardware, you will need the following components [7]:

1. ESP32 board
2. Raspberry Pi board
3. USB cable
4. Jumper wires
5. Breadboard (Optional)

Here are the steps to connect the ESP32 with Raspberry Pi in hardware:

1. Connect the ESP32 board to your computer using a USB cable [7].
2. Open a terminal window and type "ls /dev/tty*" to find the name of the serial port that your ESP32 is connected to [7].
3. Connect the GPIO pins of the ESP32 board to the GPIO pins of the Raspberry Pi board using jumper wires [7].
4. Make sure that you connect the ground pin of both boards together [7].
5. Install any necessary drivers or software on your Raspberry Pi board for communicating with the ESP32 over serial [7].
6. Open a terminal window on your Raspberry Pi and type "sudo minicom -D /dev/ttyUSB0" (replace ttyUSB0 with your actual serial port name) [7].
7. You should now be able to communicate with your ESP32 board from your Raspberry Pi [7].

Note: If you want to use a breadboard, you can connect both boards to it and then use jumper wires to connect their GPIO pins together.

That's it! You have successfully connected your ESP32 with Raspberry Pi in hardware using Embedded Linux [7].

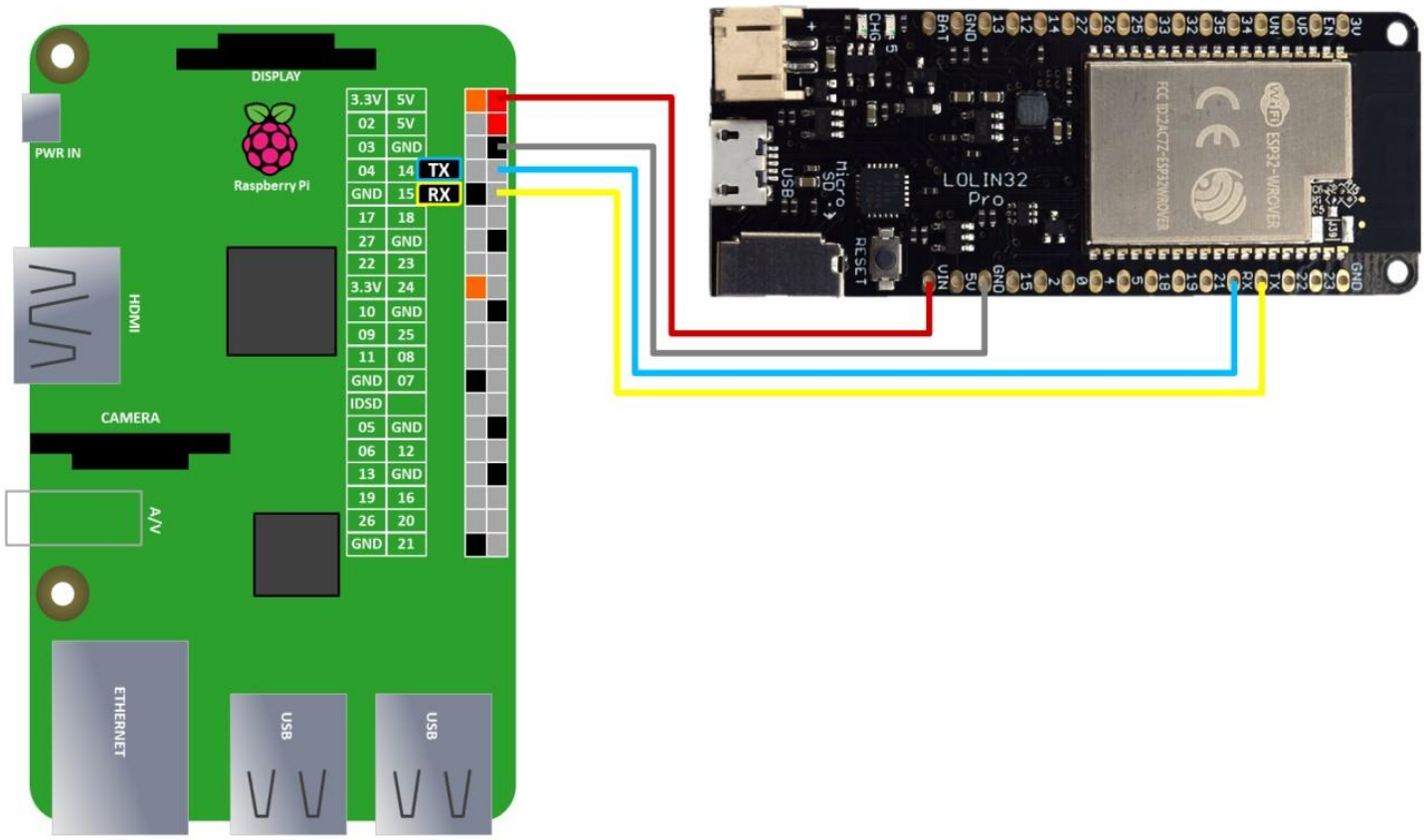


Figure 2.5 Connect the ESP32 With Raspberry Pi

2.4 Raspberry Pi with MC Using UART

We used the UART Communication Protocol. Raspberry Pi has two in-built UARTs which are as follows [8]:

- PL011 UART
- mini UART

PL011 UART is an ARM-based UART. This UART has better throughput than miniUART [8].

UART (Universal Asynchronous Receiver/Transmitter) is a serial communicationprotocol in which data is transferred serially i.e., bit by bit [8].

Asynchronous serial communication is widely used for byte-oriented transmission.In Asynchronous serial communication, a byte of data is transferred at a time [8].

UART serial communication protocol uses a defined frame structure for their data bytes [8].

Frame structure in Asynchronous communication consists of:

- START bit: It is a bit that indicates that serial communication has started, andit is always low [8].
- Data bits packet: Data bits can be packets of 5 to 9 bits. We usually use an 8-bit data packet, which is always sent after the START bit [8].
- STOP bit: This usually is one or two bits in length. It is sent after the data bitpacket to indicate the end of the frame. The stop bit is always logic high.

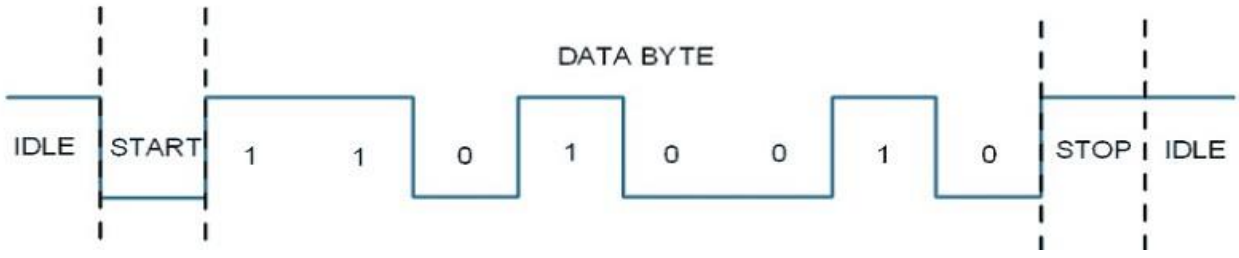


Figure 2.6 UART Frame

Usually, an asynchronous serial communication frame consists of a START bit (1 bit) followed by a data byte (8 bits) and then a STOP bit (1 bit), which forms a 10-bitframe as shown in the figure above. The frame can also consist of 2 STOP bits insteadof a single bit, and there can also be a PARITY bit after the STOP bit [8].

2.4.a Hardware Connection of Raspberry Pi with MC Using UART

1. Connect the ground (GND) pin of the Raspberry Pi to the ground pin of the microcontroller [9].
2. Connect the TX pin of the Raspberry Pi to the RX pin of the microcontroller [9].
3. Connect the RX pin of the Raspberry Pi to the TX pin of the microcontroller.
4. Power up both devices [9].
5. Enable UART on the Raspberry Pi by editing the /boot/config.txt file [9]. Add the following line at the end of the file:
   enable_uart=1
6. Install a serial communication program on the Raspberry Pi, such as minicom or screen [9].
7. Configure the serial port settings in the program to match those of your microcontroller, such as baud rate, data bits, stop bits, and parity [9].
8. Open a terminal window in your program and start communicating with your microcontroller using UART commands [9].

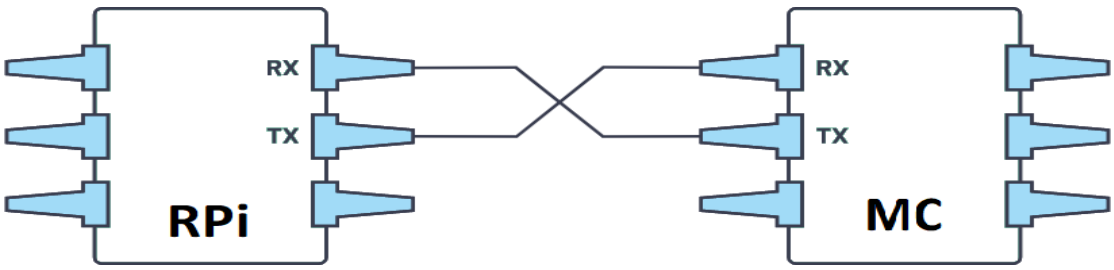


Figure 2.7 UART Connection

Sensors&Actuators 3

- The V2V Collision Avoidance system relies on a set of sensors and actuators to detect and analyze information about the surrounding vehicles and prevent collisions between them.
- These sensors and actuators are installed in the car and work together to provide a detailed picture of the environment around the vehicle.

Among the sensors that can be used in the V2V Collision Avoidance systems are:

3.1 LCD

3.1.1 LCD Theory of work

- A liquid-crystal display (LCD) is a flat panel display that uses the light modulating properties of liquid crystals [10].
- Liquid crystals do not emit light directly. 20x4 means that 20 characters can be displayed in each of the 4 rows of the 20x4 LCD, thus a total of 80 characters can be displayed at any instance of time. LCD accepts two types of signals, one is data, and another is control [10].
- These signals are recognized by the LCD module from the status of the RS pin [10].
- Now data can be read also from the LCD and Display, by pulling the R/W pin high [10].
- As soon as the E pin is pulsed, LCD display reads data at the falling edge of the pulse and executes it, same for the case of transmission [10].

3.1.2 About The Module

We used monochromatic LCD with size of 20x4 alphanumeric

- 5x8 dots.
- +5V power supply.
- 1/16 duty cycle.
- LED Backlight

Figure 3.1 LCD Module

3.1.3 LCD Pins Configuration

- Data pins D7-D0: Bi-directional data /command pins alphanumeric characters are sent in ASCII format.

- RS (register select).
  -RS = 0 → (command register).
  -RS = 1 → (data register).
- R/W (Read or Write).
  -0 (Write) /1(Read).
- E (Enable data)
  -Used to enable the LCD display.
- V0 (contrast control)
  -For maximum brightness V0=0.
- (DDRAM)
  Addresses inside LCD stores display data represented in 8-bit character codes.

• Its extended capacity is 80*8 bits or 80 characters.
• The area in Display Data RAM (DDRAM) that is not used for display Can be used as general data RAM.
• So whatever you send on the DDRAM is actually displayed on the LCD[10].

Table 3.1 Pins Configuration

Pin No.	Pin Name	Descriptions
1	VSS	Ground
2	VDD	Supply voltage for logic
3	V0	Input voltage for LCD
4	RS	H : Data signal, L Instruction signal
5	R/W	H : Read mode, L : Write mode
6	E	Chip enable signal
7	DB0	Data bit 0
8	DB1	Data bit 1
9	DB2	Data bit 2
10	DB3	Data bit 3
11	DB4	Data bit 4
12	DB5	Data bit 5
13	DB6	Data bit 6
14	DB7	Data bit 7
15	LED_A	Backlight Anode
16	LED_K	Backlight Cathode

		COLUMN									
		1st	2nd	3rd	4th	5th	.....	16th	17th	18th	19th	20th
ROW	1st	00h	01h	02h	03h	04h	.....	0Fh	10h	11h	12h	13h
	2nd	40h	41h	42h	43h	44h	20 x 4 Characters (5x8 dots font)	4Fh	50h	51h	52h	53h
	3rd	14h	15h	16h	17h	18h		23h	24h	25h	26h	27h
	4th	54h	55h	56h	57h	58h	.....	63h	64h	65h	66h	67h

DDRAM Address Map

Table 3.2 DDRAM Address Map

- **GROM Memory**

Character Generator ROM memory contains a standard character map all characters that can be displayed on the screen. Each character is assigned to one memory location.

- **LCD commands**

Command	Function
0F	LCD ON, Cursor blinking ON, Cursor ON
01	Clear screen
02	Return home
04	Decrement cursor
06	Increment cursor
0E	Cursor blinking OFF, Display ON
80	Force cursor to the beginning of 1st line
C0	Force cursor to the beginning of 2nd line
08	Display OFF, Cursor OFF
83	Cursor line 1 position 3
3C	Activate second line
38	Use 2 lines and 5x7 matrix
C1	Jump to second line, position1
OC	Cursor OFF, Display ON
C2	Jump to second line, position2

Table 3.3 LCD Commands

3.1.3 Use of LCD in our project

At first we interfaced the LCD with controller in 8 it mode, which caused problems later as the number of modules used increased [10].

To use LCD some of the above commands must be sent for initialization before sending data, so we implemented initialization function and a function to clear LCD screen. For better code organization we have 2 independent functions, one to print string variables, other to print numeric variables. Besides a function to set the cursor position to start writing at desired locations [10].

3.1.3 Optimization

In order to decrease number of GPIO pins used to interface between LCD and controller we used LCD in 4-bit mode in which data is sent in two cycles and we modified the code accordingly [10].

3.2 ULTRASONIC SENSOR

3.2.1 Ultrasonic theory of work

The ultrasonic sensor works on the principle of SONAR system which is used to determine the distance to an object. SONAR basically stands for Sound Navigation and Ranging. An ultrasonic sensor generates the high-frequency sound (ultrasound) waves. When this ultrasound hits the object, it reflects as echo which is sensed by the receiver. By measuring the time required for the echo to reach the receiver, we can calculate the distance[11].

3.2.2 About The Module

Ultrasonic module HC-SR04 has an ultrasonic transmitter, receiver, and control circuit [11].

Microcontroller U1

The heart of the unit is the EM78P153 8-bit microprocessor [11].

This handle:

- Interface to Trigger and Echo pins.
- Timing and sending antiphase burst for ping to send.
- Squelch control, whereby during Ultrasonic transmit, a threshold for the incoming receiver is effectively disabled threshold for the incoming receiver is effectively disabled to avoid bogus echoes. This is important as whilst sending vibrations from the TX transducer will be received through PCB and by air between the transducers on the RX transducer [11].

- Receiving the processed signal from the Receiver section as an interrupt (Rising edge), this is actually a filtered and much amplified version of all the echoes received[11].

Figure 3.2 HC-SR04 Ultrasonic module

Transmitter U3

- Voltage drive to TX transducer from the antiphase TX signals from the micro (U1). By using antiphase signals, a differential voltage can be driven across the transducer effectively +/-5V across the transducer [11].
- The other part is two transistors with a common base pin, collectors available on other pins. This transistor forms part of the feedback loop on the final part of the receiver chain, to change an analog signal into a TTL type digital signal [11].

Receiver U2

- The quad op-amp IC (U2) is a LM324 is 1 MHz unity gain bandwidth device, with limited range I/O. It is a ubiquitous and cheap device [11].
- Considering some of the gain levels used in the stages means that 40 kHz signals are passing through stages around 10 kHz bandwidth [11].

Operating Condition

Figure 3.3 Ultrasonic Operation condition

- HC-SR04 works at 40 KHz Frequency which lies in ultrasound range, above 20 KHz[11].
- The working voltage is 5V DC. As the current drawn by the sensor is less than 15mA, so won't be affecting the current ratings of the controller. No need for external buffers [11].

3.2.3 Working Method

- We need to transmit trigger pulse of at least 10 us to the HC-SR04 Trig Pin. 10 µs is enough periods for the controller, after which it starts to transmit ultrasonic signal [11].
- Then the HC-SR04 automatically sends Eight 40 kHz sound wave pulses and waits for rising edge output at Echo pin [11].
- As the number of pulses, the amplitude of the received signal increases and saturates at a point [11].
- After further increase in number of pulses results in increase in reflection peak, which are not required [11].
- When the rising edge capture occurs at Echo pin, start the Timer and wait for falling edge on Echo pin. As soon as the falling edge is captured at the Echo pin, read the count of the Timer [11].
- This time count is the time required by the sensor to detect an object and return back from an object [11].

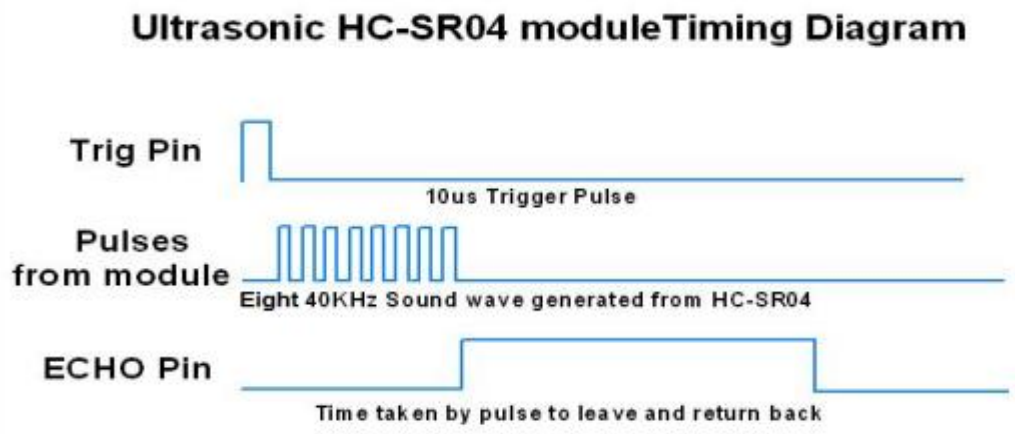

Ultrasonic HC-SR04 moduleTiming Diagram
Trig Pin
10us Trigger Pulse
Pulses from module
Eight 40KHz Sound wave generated from HC-SR04
ECHO Pin
Time taken by pulse to leave and return back

Figure 3.4 HC-SR04 modules Timing Diagram

Conversion from duration to distance can be done using the following formula:

Distance = (Travel time/2) x speed of sound

We need to divide the travel time by 2 because we have to take into account that the wave was sent, hit the object, and then returned back to the sensor [11].

If the HC-SR04 does not receive an echo, then the output never goes low. Accordingly, some sensors timeout from 28ms to 36ms [11].

3.2.4 Use of Ultrasonic in our project

In order to measure distance, we started by choosing two GPIO Pins. One for sending trigger and other for receiving echo.

Both pins are either high or low, so it was better to be used in digital mode.

We start by sending trigger to the module by setting the Trigger pin high for 10 us, and then low.

After the trigger, the module starts transmitting ultrasound wave and Echo pin in set high until the wave hits and object and return back to the module.

In order to check whether the Echo pin is high or low it was configured to receive interrupts at both rising and falling edges.

Using polling wasn't the best solution as it is time-consuming and delays other tasks.

Once the Echo pin is high, we receive a rising edge interrupt. At this instance, we reset the timer to measure the duration in which Echo pin is high.

When the Echo pin returns low, we receive a falling edge interrupt, disable the timer and calculate the distance.

Since the operating frequency of controller is 80 MHz, so by diving number of ticks by 80 we get the

Duration at which Echo pin is high in µs.

Assuming the speed of transmitted ultrasound wave is **approximately** 343 m/s, so 343m/s = 0.0343 cm/µs = 1/29.1 cm/µs.

Then we divide it by 2 to get the required distance in cm.

To provide 360° awareness for the car and avoid crashes, position and number of sensors had to be taken into consideration. We used 6 Ultrasonic sensors to measure distance between cars in all directions, including blind spots [11].

```
if (!Echo)
  {
    TIMER2_TAV_R  = 0;
    TIMER2_CTL_R= 0x00000001;
    Echo=1;
  }

  else
   {
    pulse = TIMER2_TAR_R;
    TIMER2_CTL_R = 0x00000000;
    Echo=0;
    Reading =(unsigned long)(pulse/80.0);
    Reading = Reading / 58;
```

As the number of sensors increased, we had some constraints such as: Number of GPIO pins and Timers [11].

• **GPIO**

- To use less number of GPIO pins, we thought of using only one echo pin for all sensors, but it wasn't the best solution as the controller receive interrupts from more than one sensor at the same time, this problem was slightly solved using a delay function but not for more than three sensors and there was a great inference between sensors readings [11].
- Besides, increasing delay between triggering sensors was time consuming and increased task duration [11].
- For improved operation, we connected all sensors to the same GPIO port so that interrupts can be easily enabled and disabled at once if needed [11].
- To ensure we get correct reading before sending trigger to a sensor we enable interrupts on the equivalent Echo pin and disable interrupts on all other Echo pins, to evade simultaneous interrupts [11].

• **Timers**

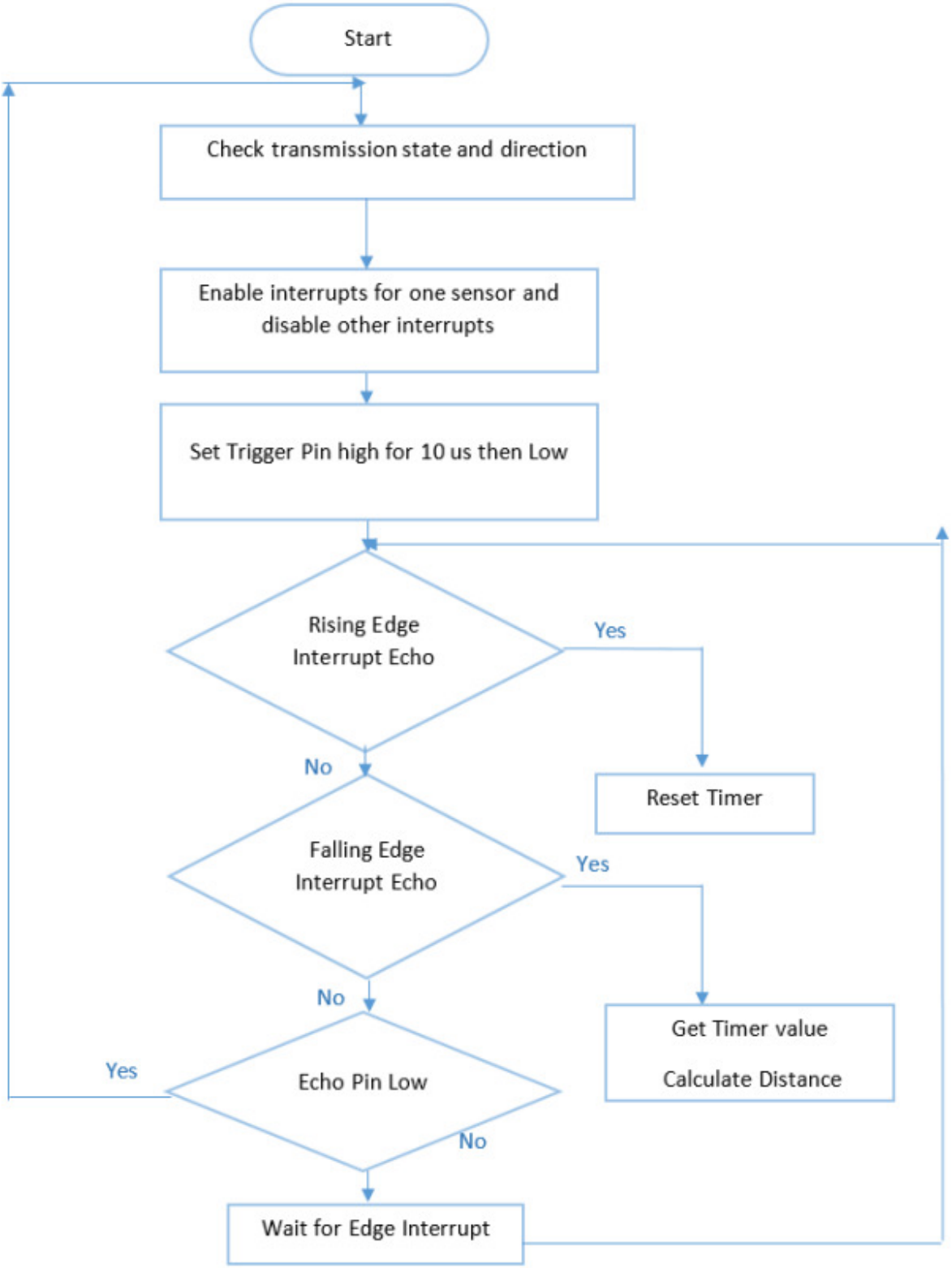
- We had to use 32-bit timer to avoid timer overflow and wrong results, at operating frequency of 80 MHz and by using 16 bit timer, the maximum number of counts = 2^{16}= 65536, Then time = $\frac{65536}{80*10^6}$ = 0.0008192, so maximum distance is approximately 281 cm which is less than ultrasonic operating range. Thus, it was better to use 32-bit timer [11].
- We thought of using 6 independent timers, one for each sensor, but this solution had some problems as by sending trigger to multiple sensors at the same time you can get multiple interrupts at the same time from the various echo signals, which caused inaccurate values [11].
- To avoid wrong readings we agreed on using only one timer, sending trigger to one sensor at a time and loop on other sensors [11].

3.2.5 Optimization

Since we have 6 Ultrasonic sensors, we can consider this case. The car is moving in a forward direction and there is another one in front of it. Here, we are interested in the front sensor reading as it is the most important. Another case, if the driver wants to turn right, so the right sensor's reading is the most important.

Same for turning left or moving backward. Accordingly, the code was optimized in a way such that the order we get reading from ultrasonic sensor changes according to direction and transmission state. As a result, the car has become more reliable and more responsive to obstacles. Also, we had better control for motors' speed [11].

3.2.6 Ultrasonic Flowchart



3.3 DC MOTOR

3.3.1 About the module

- Most prominently, these areelectric motors or pneumatic actuators with electric valves[12].
- We will concentrate on electric actuators using Direct Current (DC) [12].
- Single-phase DC electric motors are arguably the most used method for locomotion in mobile robots[12].
- DC motors are clean, quiet, and can produce sufficient power for a variety of task[12].
- They are much easier to control than pneumatic actuators, which arefrequently used when high torques are required[12].
- The first step selection the appropriate motorsystem[12].
- The best choice is an encapsulated motor combination comprising:
  - DC motor
  - Gearbox
  - Optical or magnetic encoder (dual phase-shifted encoders for detection of speedand direction).

Using encapsulated motor systems has the advantage that the solution is much smaller than those using separate modules, plus the system is dust-proof and shielded against stray light, which is required for optical encoders. The disadvantage of using a fixedassembly like this is that the gear ratio may only be changed with difficulty, or not atall. A magnetic encoder comprises a disk equipped with several magnets and a Hall-effective[12].

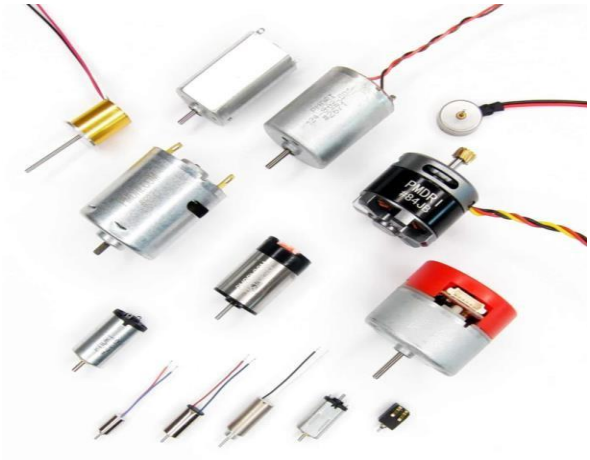


Figure 3.5 DC Motors Types.

3.3.2 About the H-Bridge

H Bridge is a simple electronic circuit that enables us to apply voltage to load in either direction. It is commonly used to control DC Motors. By using H Bridge, we can run DC Motor in clockwise or anticlockwise directions. Most mini-DC motors have two terminals because they are series type (armature and fieldwindings are connected in series) DC motors [12].

If we connect terminal A with +Ve supply and terminal B with –Ve supply or ground, the current will flow from the motor from A to B and the motor will rotate in one direction – say clockwise (CW) or forward direction. By changing the supply terminals [12].

Now B relates to +Ve and A is connected to the ground. The current will flow from the motor from B to A and the motor will rotate in another direction (counterclockwise – CCW or reverse).

Four switches are connected in between +Ve supply and ground and the DC motor isconnected in between two switches as shown [12].
Such circuit arrangement is known asH-bridge because it looks like the letter 'H'. If SW1 and SW4 are pressed simultaneously then current will flow from +Ve – SW1 – A – B – SW4 – Gnd. So, the motor will rotate in one direction.

Now if SW2 and SW3 are pressed current will flow from +Ve – SW2 – B – A – SW3– Gnd. So, the motor gets reverse supply, and it will rotate in another direction. Thecircuit replaces the switches with NPN-type transistors [12].

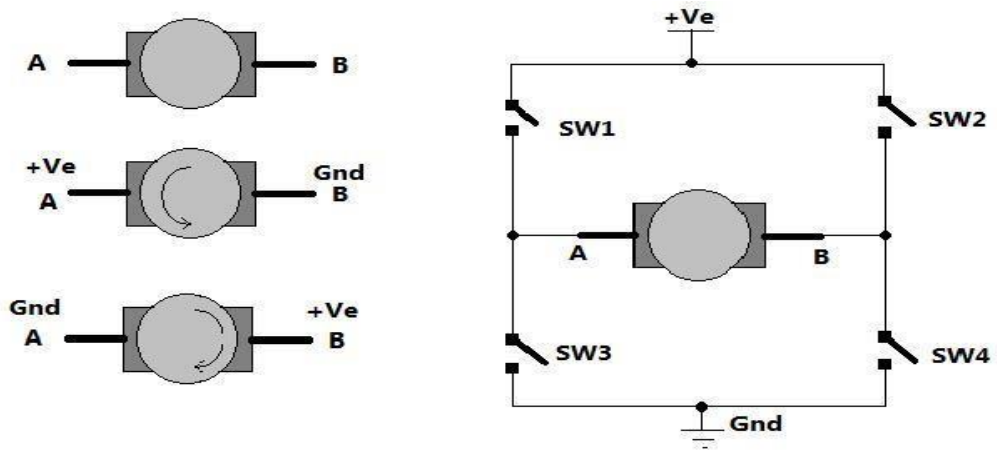


Figure 3.6 DC Motors H-Bridge.

We know that a transistorworks as a switch.
For an NPN transistor if we give +Ve input to the base it will turnON and if we give 0 input it will be turned OFF. So, in this circuit, if Q1 and Q4 are turned ON simultaneously the motor will rotate forward and if Q2 and Q3 are turnedON then the motor will rotate reverse. Motor can be rotated forward and reverse using only 4 transistors [12].

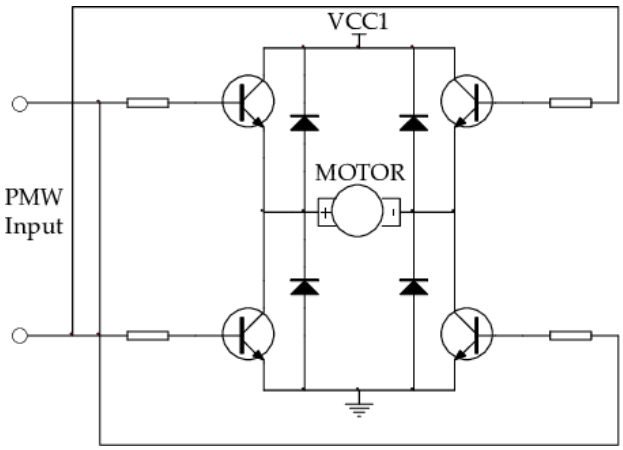


Figure 3.7 H-Bridge Circuit

3.3.3 Working Method

- The DC motor speed varies as the applied input voltage varies. As you increase the applied input voltage the speed will increase and vice versa [12].
- Applying max rated voltage will rotate the motor at full speed [12].
- One of the very popular methods for generating variable DC voltage is pulse width modulation (PWM) [12].
- Pulse width modulation means varying the width (duty) of the pulse. Width means ON time Ton of pulse [12].

- **<u>The average output voltage (Vdc or Vave) is given by the equation:</u>**

  - $Vdc = \frac{T_{on}}{T_{on} + T_{off}} * V_S$
  - It directly depends upon Ton.

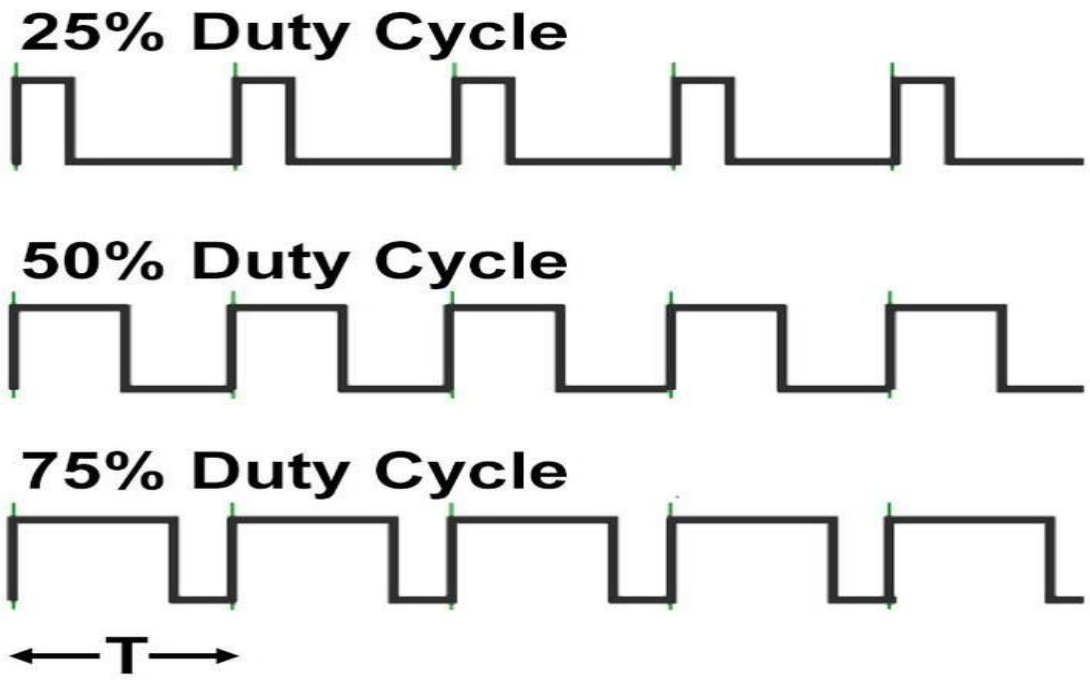


Figure 3.8 PWM Duty cycle

- If duty is 50% the average output voltage Vdc is exactly half of Vs[12].
- If duty is increased to 75%, Vdc also increases to 3/4th of Vs and if duty is decreased to 25%, Vdc reduces to 1/4th of Vs. Thus, as pulse width varies the average output voltage varies [12].
- So, we must apply PWM to the DC motor to vary its Speed [12].

3.4 HMC5883L Sensor

The HMC5883L is a 3-axis digital compass sensor developed by Honeywell. It is commonly used in electronic devices such as smartphones, tablets, and GPS systems to provide heading information [13].

Here are some key features of the HMC5883L:

- 3-axis magneto resistive sensor.
- Digital output via I2C interface.
- 12-bit resolution for each axis.
- Adjustable output data rate up to 75 Hz.
- Built-in self-test capability.
- Low power consumption (100 µA at 3V).

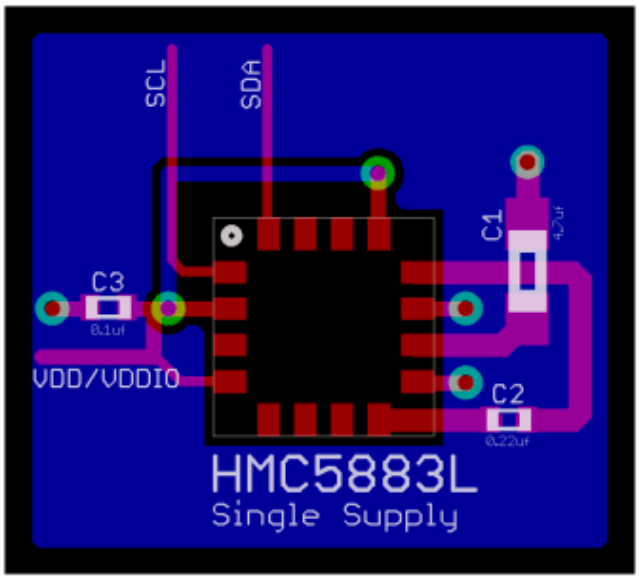


Figure 3.9 HMC5883L single supply

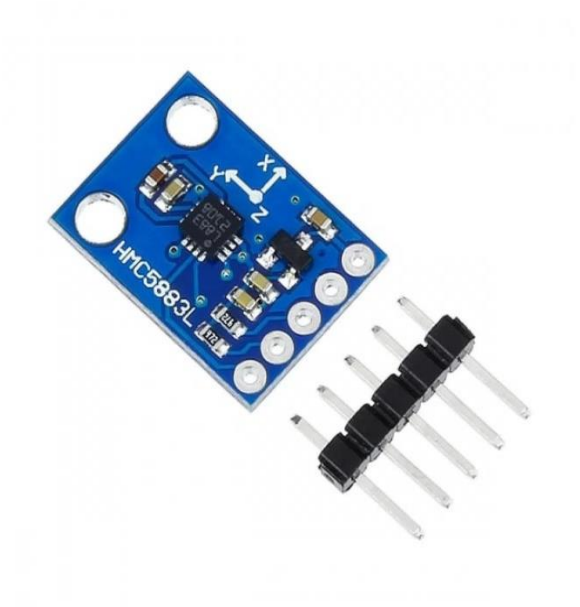


Figure 3.10 HMC5883L module

3.4.1 About the module

PIN CONFIGURATIONS

Pin	Name	Description
1	SCL	Serial Clock – I^2C Master/Slave Clock
2	VDD	Power Supply (2.16V to 3.6V)
3	NC	Not to be Connected
4	S1	Tie to VDDIO
5	NC	Not to be Connected
6	NC	Not to be Connected
7	NC	Not to be Connected
8	SETP	Set/Reset Strap Positive – S/R Capacitor (C2) Connection
9	GND	Supply Ground
10	C1	Reservoir Capacitor (C1) Connection
11	GND	Supply Ground
12	SETC	S/R Capacitor (C2) Connection – Driver Side
13	VDDIO	IO Power Supply (1.71V to VDD)
14	NC	Not to be Connected
15	DRDY	Data Ready, Interrupt Pin. Internally pulled high. Optional connection. Low for 250 µsec when data is placed in the data output registers.
16	SDA	Serial Data – I^2C Master/Slave Data

Table 3.4 HMC5883L Pins Configuration

3.4.2 HMC5883L theory of work

The HMC5883L is a 3-axis digital compass sensor that uses magneto resistive sensors to measure the Earth's magnetic field in three orthogonal axes (X, Y, and Z). The sensor is based on the principle of magnetoresistance, which is the change in resistance of a material when exposed to an external magnetic field[13].

The HMC5883L sensor consists of three magneto resistive sensors aligned in orthogonal directions, which allows it to measure the magnetic field in three dimensions. The sensor output is digitized and provided through an I2C interface [13].

To measure the magnetic field, the HMC5883L sensor uses a technique called "flux gate magnetometer" [13].

This involves using a magnetic core that is driven into saturation by an AC signal. The magnetic core is then exposed to the external magnetic field, which causes a change in the core's magnetic permeability [13].

This change in permeability is detected by the magneto resistive sensors, which then provide an output voltage proportional to the magnetic field strength [13].

The output of the HMC5883L sensor is a digital value that represents the magnetic field in each of the three axes. The output data can be processed to provide heading information in degrees or radians [13].

To ensure accurate measurements, the HMC5883L sensor must be properly calibrated. Calibration involves measuring the sensor's output in various orientations and applying correction factors to compensate for any errors caused by hard iron and soft iron distortions in the magnetic field [13].

Overall, the HMC5883L sensor provides a reliable and accurate method for measuring magnetic fields and determining heading information. [13].

3.4.3 Use of HMC5883L in our project

The HMC5883L sensor can contribute to V2V collision avoidance in the following ways:

1. Providing accurate heading information: The HMC5883L sensor can be used to provide accurate heading information, which is critical for determining the direction of nearby vehicles and predicting their movements [13].
2. Detecting magnetic fields: The HMC5883L sensor can detect the magnetic fields generated by nearby vehicles, which can be used to provide additional information about their position and movement [13].
3. Improving accuracy of other sensors: The HMC5883L sensor can be used in conjunction with other sensors, such as radar and lidar, to improve their accuracy and provide a more complete picture of the surrounding environment [13].
4. Reducing false alarms: The HMC5883L sensor can be used to filter out false alarms caused by other sources of electromagnetic interference, such as power lines and electrical equipment [13].

Overall, the HMC5883L sensor can make an important contribution to V2V collision avoidance systems by providing accurate and reliable information about the position and movement of nearby vehicles. By improving the accuracy of these systems, the HMC5883L sensor can help to reduce the risk of accidents and improve safety on the roads [13].

3.5 BUZZER

- An audio signaling device like a beeper or buzzer may be electromechanical or piezoelectric or mechanical type [14].
- The main function of this is to convert the signal from audio to sound. Generally, it is powered through DC voltage and used in timers, alarm devices, printers, alarms, computers, etc [14].
- Based on the various designs, it can generate different sounds like alarm, music, bell & siren [14].

3.5.1 Buzzer Pin Configuration

The **pin configuration of the buzzer** is shown below. It includes two pins namely positive and negative [14].

The positive terminal of this is represented with the '+' symbol or a longer terminal. This terminal is powered through 6Volts whereas the negative terminal is represented with the

'-' symbol or short terminal and it is connected to the GND terminal [14].

Figure 3.11 BUZZER Module

3.5.2 Specifications

The **specifications of the buzzer** include the following.

- Color is black.
- The frequency range is 3,300Hz.
- Operating Temperature ranges from – 20° C to +60°C.
- Operating voltage ranges from 3V to 24V DC.
- The sound pressure level is 85dBA or 10cm.
- The supply current is below 15mA.

3.5.3 Working principle

The working principle of a buzzer is based on **the piezoelectric effect**.

- A piezoelectric material is one that generates an electric charge in response to mechanical stress and produces mechanical stress in response to an electric field. Buzzer sensors typically use a piezoelectric ceramic element as the sound-producing element [14].
- When a voltage is applied across the piezoelectric element, it causes the element to vibrate at a high frequency, typically in the range of several kilohertz to several megahertz [14].
- These vibrations cause the air molecules around the element to vibrate, producing sound waves that propagate through the air [14].
- The frequency of the sound produced by the buzzer is determined by the frequency of the voltage applied to the piezoelectric element [14].
- The intensity of the sound, or the volume, is determined by the amplitude of the voltage [14].
- Buzzer sensors can be designed to produce different types of sounds, including continuous tones, intermittent tones, and complex tone patterns.
- The specific sound produced by the buzzer is determined by the design of the piezoelectric element and the electrical circuit driving it[14].
- Overall, the working principle of a buzzer is based on the piezoelectric effect and the vibration of a piezoelectric element to produce sound waves in the surrounding air [14].

3.5.4 Buzzer Circuit Diagram

The **circuit diagram of the water level indicator using the buzzer** is shown below [14].
This circuit is used to sense or detect the water level within the tank or washing machine or pool, etc [14].
This circuit is very simple to design using a few components such as a transistor, buzzer, 300K variable resistor, and power supply or 9V battery [14].

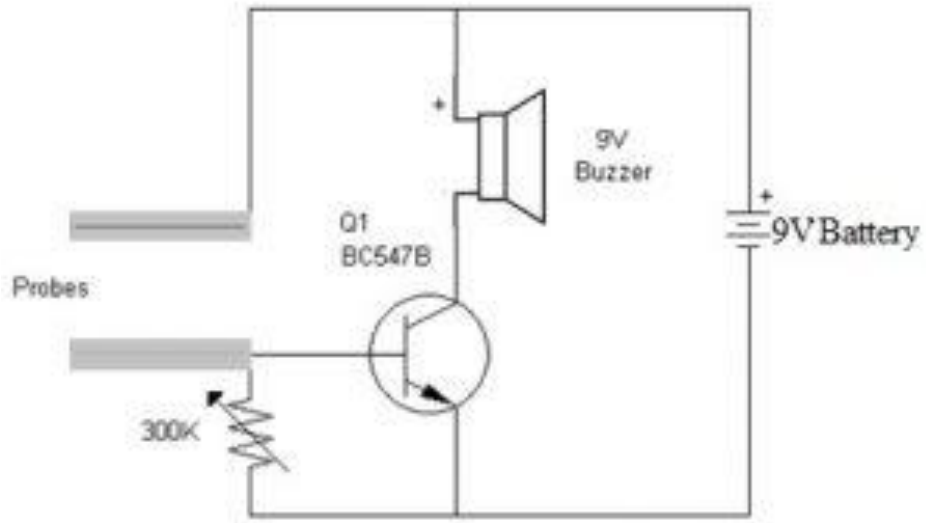


Figure3. 12 Water Level Circuit using Buzzer.

- Once the two probes of the circuit are placed in the tank, it detects the level of water [14].
- Once the water level exceeds the fixed level, it generates a beep sound through a buzzer connected to the circuit [14].

- This circuit uses a BC547B NPN transistor however we can also use any general-purpose transistor instead of using 2N3904/2N2222 [14].

- This water level sensor circuit working is very simple, and the transistor used within the circuit works as a switch [14].

- Once the two probes notice the water level within the tank, then the transistor turns ON & the voltage begins flowing throughout the transistor to trigger the buzzer [14].

B

SOFTWARE ARCHITECTURE

CONTENT :

Chapter 4: ESP32 Communication (ESP NOW)
Chapter 5: Universal Asynchronous Receiver Transmitter (UART)
Chapter 6: Bluetooth
Chapter 7: Communication between ESP32 and Raspberry Pi

ESP NOW 4

In this chapter, we'll talk about ESP NOW in general, a connectionless protocol developed by Espressif for the ESP32 and ESP08266 microcontrollers.

With ESP-NOW we can build a private network without WIFI or a Router.

4.1 Introduction

The ESP32 and its cousin, the ESP8266, are undoubtedly remarkable microcontrollers. Aside from high-speed 32-bit architecture, they also have built-in Bluetooth and WIFI [15].

The Bluetooth and WIFI capabilities on these devices are made possible by an integrated 2.4GHz radio transceiver module [15]. This module can also be used for other communications applications that use the unlicensed 2.4GHz band [15].

Espressif, are the manufacturers of the ESP8266 and ESP32, have developed a protocol that allows all these devices to create a private, wireless network using the 2.5GHz transceivers [15].

This is a separate network from the WIFI network and can only be used by ESP-type microcontrollers. The protocol is called ESP-NOW [15].

4.2 ESP NOW

ESP NOW allows simple packet communications between ESP devices, using the 2.4 GHz band. These transmissions operate a lot like those used by wireless mice and keypads and are limited to packets of 250 bytes or fewer [16].

The proper antennas) you can achieve over 400 meters. Just using the built-in antennas data can be unidirectional or bidirectional, single-duplex or full-duplex. Most data types are supported [16].

Data can be encrypted or unencrypted, and no external source of WIFI or a router is required [16].
Depending upon your configuration, you can have anywhere from 2 to 20 devices communicating between themselves [16].

The range can vary dramatically due to the environment, but under the right conditions (and with on the modules should still allow you to communicate through a medium-sized home without a problem [16].

Figure 4.1 ESP Boards

4.2.1 ESP NOW Networking Modes

You can place your ESP-NOW network in many configurations. You can mix and match ESP32 and ESP8266 devices within the same network [16].

4.2.1.a Initiators and Responders

A device participating in an ESP-NOW network can be operated in one of two modes [16].

- **Initiator** – This device initiates the transmission. It will require the MAC address of the receiving device [16].
- **Responder** – This device receives the transmission [16].

4.2.1.b One-Way Communication

The simplest communications topology is one-way, unidirectional communications [16].

In this arrangement, the Initiator ESP32 transmits data to the Responder ESP32, the Initiator can tell if the Responder received the message successfully [16].

This is a simple arrangement, but it has many uses in remote control applications.

4.2.1.c One Initiator & Multiple Responders

This setup consists of one Initiator that is communicating with multiple responders [16].

The configuration can be used in two fashions:

- The Initiator communicates with each Responder individually [16].
- The Initiator initiates a broadcast message to communicate with all the Responders [16].

An alarm system might use this sort of configuration to activate remote sounders or communicate with remote monitors when an alarm has been triggered [16].

4.2.1.d One Responder & Multiple Initiators

This is the reverse of the previous ESP-NOW network configuration. In this arrangement, we have one Responder and multiple Initiators [16].

This arrangement is very common as it is used for remote sensor applications, with the Responder gathering data sent by the Initiator [16].

4.2.1.f Two-Way Communications

The ESP-NOW protocol can also handle bidirectional, or full-duplex, communications [16].

This illustrates the simplest arrangement, with two devices communicating with one another [16].

In this, and all bidirectional communications configurations, the ESP-32 acts as both Initiator and Resolver [16].

4.2.1.g Two-Way Networking

Expanding upon the previous configuration even further, we come up with this arrangement, four boards that have bidirectional communications established with one another [16].

The following Figures illustrate Networking modes [16].

Figure 4.2 One Way Communication.

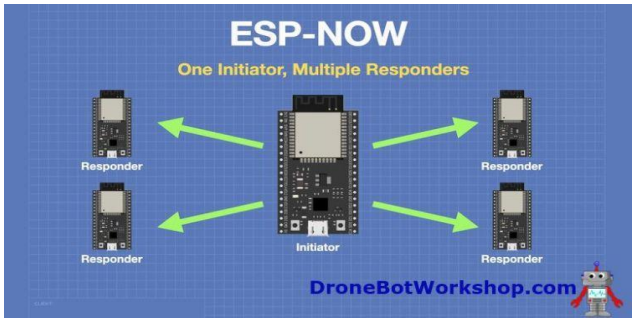


Figure 4.3 One Initiator Multiple Responders

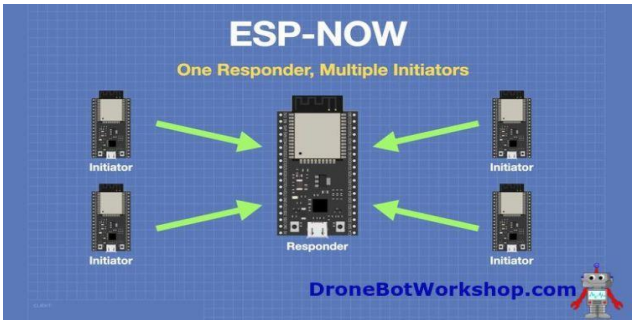


Figure 4.4 One Responder, Multiple Initiators

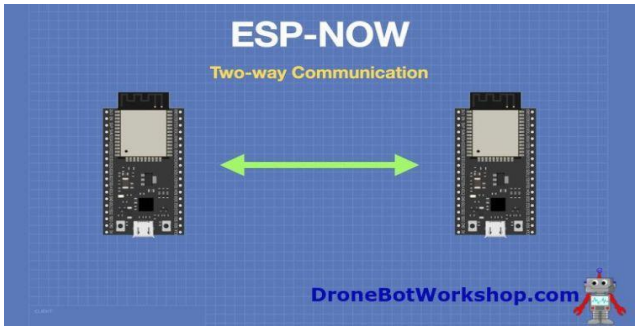


Figure 4.5 Two-Way Communication

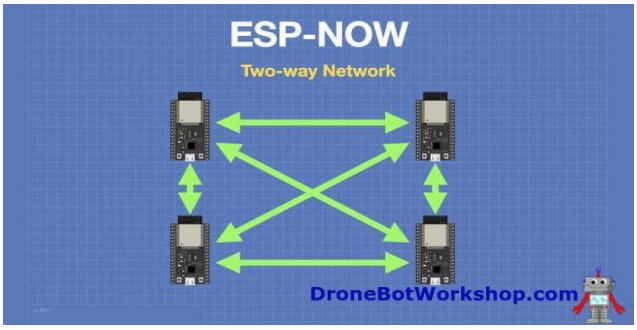


Figure 4.6 Two-Way Network

4.2.2 MAC Addresses

When Initiators communicate with Responders, they need to know the Responder's MAC Address [16].

A MAC, or Media Access Control, Address is a unique 6-digit hexadecimal number assigned to every device on a network [16].

It is generally burned into the device by the manufacturer, although it is possible to manually set it [16].

Every ESP32 and ESP8266 has its own unique MAC address, and you'll need to know that address to use it as a Responder in the experiments we will be doing today [16].

Figure 4.7 MAC Address

4.2.3 MAC Address Sketch

Here is a very simple sketch that you can run on an ESP32 to determine its unique MAC Address [16]:

```
  ESP32 MAC Address printout
  esp32-mac-address.ino
  Prints MAC Address to Serial Monitor

  DroneBot Workshop 2022
  https://dronebotworkshop.com
*/

// Include WiFi Library
#include "WiFi.h"

void setup() {

  // Setup Serial Monitor
  Serial.begin(115200);

  // Put ESP32 into Station mode
  WiFi.mode(WIFI_MODE_STA);

  // Print MAC Address to Serial monitor
  Serial.print("MAC Address: ");
  Serial.println(WiFi.macAddress());
}

void loop() {

}
```

4.2.3.a Running the MAC Sketch

The entire sketch runs in the Setup section, so after loading it to the ESP32, it will likely run before you get a chance to view it on the Serial monitor [16].

You can press the Reset key on your module to force it to run again [16].

The MAC Address will be at the bottom of the screen. Copy it to a safe location, so that you can use it later [16].

4.3 Coding for ESP-NOW

The ESP-NOW Library is included in your ESP device boards manager installation. It has several functions and methods to assist with coding for ESP-NOW [16].

In order to see how it works, you need to examine the sending and receiving of an ESP-NOW message packet [16].

4.3.1 Callbacks

A callback is a bit like an interruption, it is generated every time a specific event has occurred [16].
In the ESP-NOW protocol, there are two callbacks of interest:

- Sending Data – The **esp_now_register_send_cb ()** callback is called whenever data is sent [16].
- Receiving Data – The **esp_now_register_rcv_cb ()** callback is called when data is received [16].

In your code, you will create a callback function that you will bind to either the sending or receiving callback using the functions listed above [16].
Your function will run every time the event occurs [16].
The callback functions also return some useful data:

- The Sending callback returns the status of the sent data [16].
- The Receiving callback includes the received data packet [16].

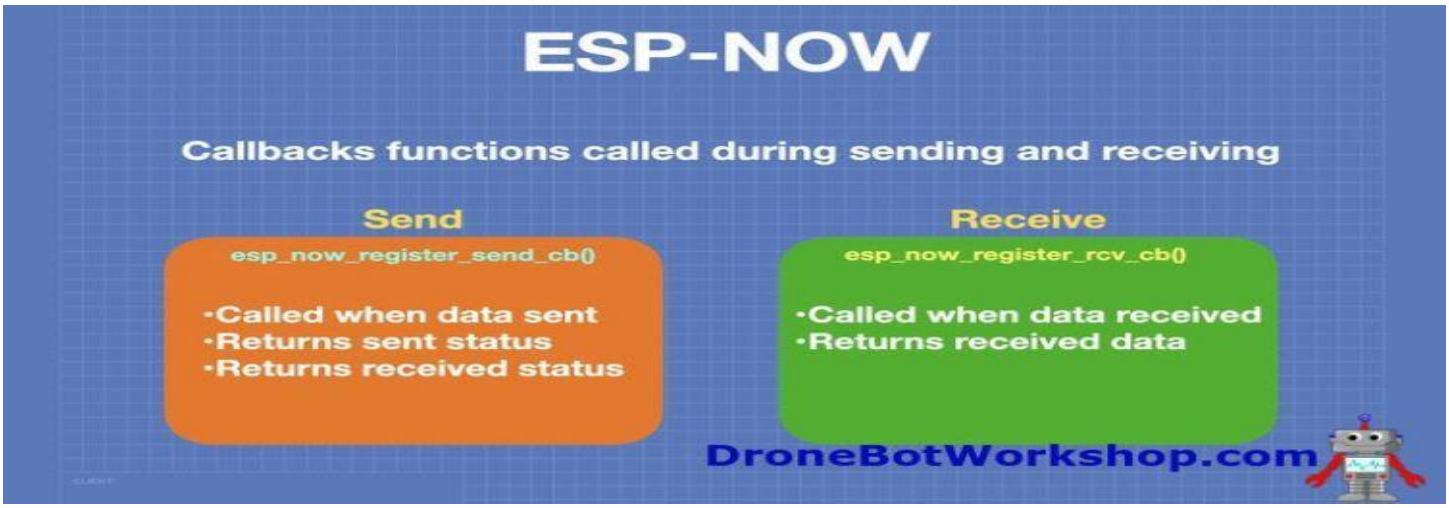


Figure 4.8 Callback Functions

4.3.2 Coding for ESP-NOW – Sending

If you are writing code to use the ESP32 or ESP8266 as the Initiator, then this is what you need to accomplish:

- You need to initialize the ESP-NOW library.
- Next, you'll register your send callback function.
- You need to add the responder device by its MAC address.
- Finally, you can packetize and send the message [16].

4.3.3 Coding for ESP-NOW – Receiving

To write code for the ESP-NOW responder, you'll need to do the following:

- You need to initialize the ESP-NOW library [16].
- Next, you'll register your receive callback function [16].
- In the receive callback, you'll capture the incoming message data and pass it to a variable [16].

Let's dig out some ESP boards and start experimenting with ESP-NOW [16].

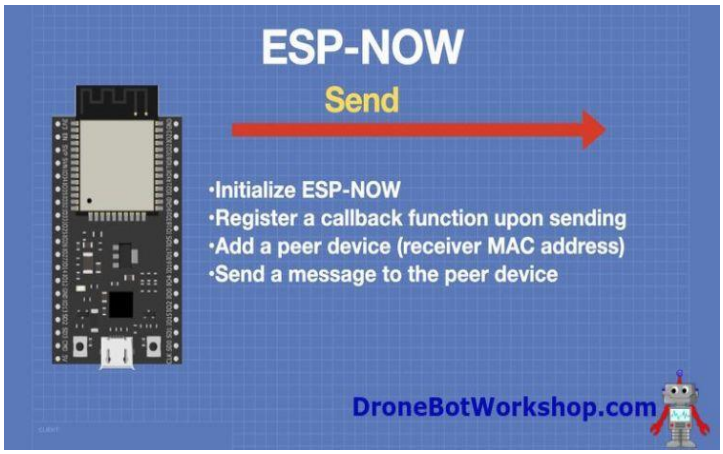


Figure 4.9 Send Process

Figure 4.10 Receive Process

4.4 More Information About ESP NOW

4.4.1 Frame Format

ESP-NOW uses a vendor-specific action frame to transmit ESP-NOW data. The default ESP-NOW bit rate is 1 Mbps [17].

The format of the vendor-specific action frame is as follows:

MAC Header	Category Code	Organization Identifier
24 bytes	1byte	3bytes
Random Values	Vendor Specific Content	FCS
4bytes	7~257bytes	4bytes

- Category Code: The Category Code field is set to the value (127) indicating the vendor-specific category [17].
- Organization Identifier: The Organization Identifier contains a unique identifier (0x18fe34), which is the first three bytes of MAC address applied by Espressif [17].
- Random Value: The Random Value filed is used to prevents relay attacks [17].
- Vendor Specific Content: The Vendor Specific Content contains vendor-specific fields as follows [17]:

Element ID	Length	Organization Identifier	Type	Version	Body
1 byte	1 byte	1 byte	1 byte	1 byte	0~250 byte

- Element ID: The Element ID field is set to the value (221), indicating the vendor-specific element [17].
- Length: The length is the total length of Organization Identifier, Type, Version and Body [17].
- Organization Identifier: The Organization Identifier contains a unique identifier(0x18fe34), which is the first three bytes of MAC address applied by Espressif [17].
- Type: The Type field is set to the value (4) indicating ESP-NOW [17].
- Version: The Version field is set to the version of ESP-NOW [17].
- Body: The Body contains the ESP-NOW data [17].

As ESP-NOW is connectionless, the MAC header is a little different from that of standard frames [17].

The From DS and To DS bits of Frame Control field are both 0. The first address field is set to the destination address [17].

The second address field is set to the source address. The third address field is set to broadcast address (0xff:0xff:0xff:0xff:0xff:0xff) [17].

4.4.2 Security

ESP-NOW uses the CCMP method, which is described in IEEE Std. 802.11-2012, to protect the vendor-specific action frame [17].

The Wi-Fi device maintains a Primary Master Key (PMK) and several Local Master Keys (LMK). The lengths of both PMK and LMk are 16 bytes [17].

- PMK is used to encrypt LMK with the AES-128 algorithm. Call esp_now_set_pmk () to set PMK. If PMK is not set, a default PMK will be used [17].

- LMK of the paired device is used to encrypt the vendor-specific action frame with the CCMP method. The maximum number of different LMKs is six [17].

4.4.3 Initialization and De-initialization

Call **esp_now_init()** to initialize ESP-NOW and **esp_now_deinit()** to de-initialize ESP-NOW. ESP-NOW data must be transmitted after Wi-Fi is started, so it is recommended to start Wi-Fi before initializing ESP-NOW and stop Wi-Fi after de-initializing ESP-NOW [17].

When **esp_now_deinit()** is called, all of the information of paired devices will be deleted [17].

4.4.4 Add Paired Device

Call **esp_now_add_peer()** to add the device to the paired device list before you send data to this device. If security is enabled, the LMK must be set [17].

You can send ESP-NOW data via both the Station and the Soft AP interface. Make sure that the interface is enabled before sending ESP-NOW data [17].

The maximum number of paired devices is 20, and the paired encryption devices are no more than 17, the default is 7 [17].

If you want to change the number of paired encryption devices, set CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM in the Wi-Fi component configuration menu [17].

A device with a broadcast MAC address must be added before sending broadcast data. The range of the channel of paired devices is from 0 to 14 [17].

If the channel is set to 0, data will be sent on the current channel. Otherwise, the channel must be set as the channel that the local device is on [17].

4.4.5 Send ESP-NOW Data

Call **esp_now_send()** to send data by using ESP-NOW library then we have to use **esp_now_register_send_cb()** to register sending callback function [17].

It will return *ESP_NOW_SEND_SUCCESS* in sending callback function if the data is received successfully on the MAC layer [17].

Otherwise, it will return *ESP_NOW_SEND_FAIL*. Several reasons can lead to ESP-NOW fails to send data. For example, the destination device doesn't exist; the channels of the devices are not the same; the action frame is lost when transmitting on the air, etc. It is not guaranteed that the application layer can receive the data [17].

If necessary, send back ack data when receiving ESP-NOW data [17].

If receiving ack data timeouts, retransmit the ESP-NOW data [17].

A sequence number can also be assigned to ESP-NOW data to drop the duplicate data [17].

If there is a lot of ESP-NOW data to send, call **esp_now_send()** to send less than or equal to 250 bytes of data once a time [17].

Note that too short interval between sending two ESP-NOW data may lead to disorder of sending callback function [17].

So, it is recommended that sending the next ESP-NOW data after the sending callback function of the previous sending has returned [17].

The sending callback function runs from a high-priority Wi-Fi task [17].

So, do not do lengthy operations in the callback function [17].

Instead, post the necessary data to a queue and handle it from a lower priority task [17].

4.4.6 Receiving ESP-NOW Data

Call **esp_now_register_recv_cb()** to register receiving callback function. Call the receiving callback function when receiving ESP-NOW [17].

The receiving callback function also runs from the Wi-Fi task [17].

So, do not do lengthy operations in the callback function [17].

Instead, post the necessary data to a queue and handle it from a lower priority task [17].

4.4.7 Config ESP-NOW Rate

Call **esp_wifi_config_espnow_rate()** to config ESPNOW rate of specified interface. Make sure that the interface is enabled before config rate [17].

This API should be called after **esp_wifi_start()** [17].

4.4.8 Config ESP-NOW Power-saving Parameter

Sleep is supported only when ESP32 is configured as station [17].

Call **esp_now_set_wake_window()** to configure Window for ESP-NOW RX at sleep. The default value is the maximum, which allows RX all the time.

If Power-saving is needed for ESP-NOW [17]: Call the following API to configure enable power saver mode and set its interval duration **esp_wifi_connectionless_module_set_wake_interval().**

UART

5

In this chapter, we'll talk about Universal Asynchronous Receiver/Transmitter protocol (UART) in general.

Advantages, disadvantages, how it works, the hardware interface, data, protocol frames and UART operation.

5.1 Introduction

UART stands for Universal Asynchronous Receiver/Transmitter. It's not a communication protocol like SPI and I2C, but a physical circuit in a microcontroller, or a stand-alone IC. A UART's main purpose is to transmit and receive serial data [18].

In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device [18].

Only two wires are needed to transmit data between two UARTs.

Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART [18].

UARTs transmit data *asynchronously*, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART [18].

Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet, so the receiving UART knows when to start reading the bits.

When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the *baud rate*. Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps) [18].

Both UARTs must operate at about the same baud rate. The baud rate between the transmitting and receiving UARTs can only differ by about 10% before the timing of bits gets too far off [18].

5.2 UART Interfacing

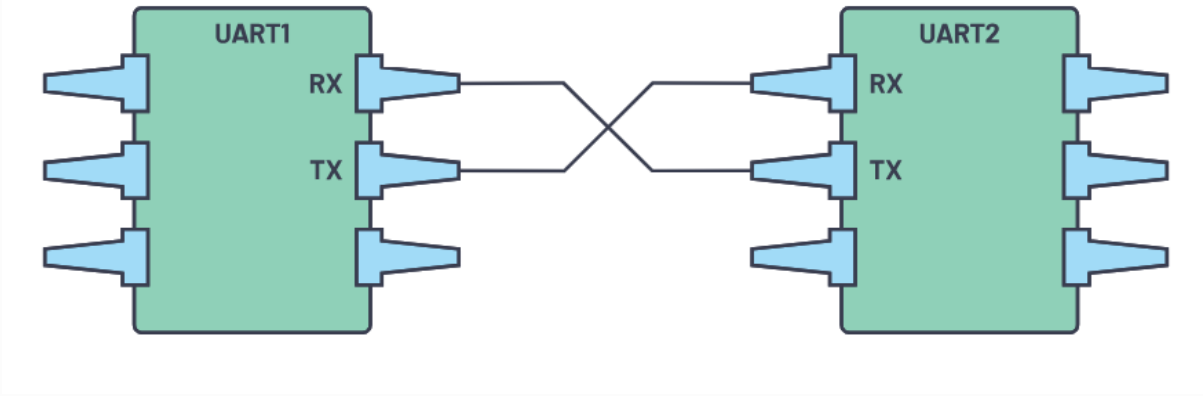


Figure 5.1 Two UARTs communicate directly with each other.

The two signals of each UART device are named:

- Transmitter (Tx)
- Receiver (Rx)

The main purpose of a transmitter and receiver line for each device is to transmit and receive serial data intended for serial communication [18].

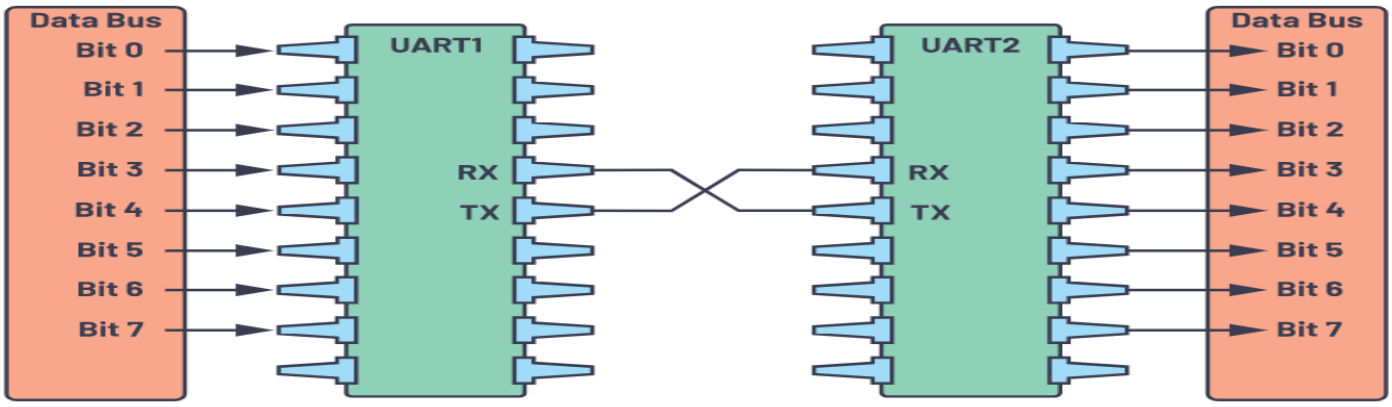


Figure 5.2 *UART with data bus*.

The transmitting UART is connected to a controlling data bus that sends data in a parallel form [18].

From this, the data will now be transmitted on the transmission line (wire) serially, bit by bit, to the receiving UART [18].

This, in turn, will convert the serial data into parallel for the receiving device [18].

The UART lines serve as the communication medium to transmit and receive one data to another [18].

Take note that a UART device has a transmit and receive pin dedicated for either transmitting or receiving [18].

For UART and most serial communications, the baud rate needs to be set the same on both the transmitting and receiving device [18].

The baud rate is the rate at which information is transferred to a communication channel [18].

In the serial port context, the set baud rate will serve as the maximum number of bits per second to be transferred [18].

Table 5.1 summarizes what we must know about UART.

Wires	2
Speed	9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600, 1000000, 1500000
Methods of Transmission	Asynchronous
Maximum Number of Masters	1
Maximum Number of Slaves	1

Table 5.1 UART Summary

The UART interface does not use a clock signal to synchronize the transmitter and receiver devices; it transmits data asynchronously [18].

Instead of a clock signal, the transmitter generates a bitstream based on its clock signal while the receiver is using its internal clock signal to sample the incoming data [18].

The point of synchronization is managed by having the same baud rate on both devices [18].

Failure to do so may affect the timing of sending and receiving data that can cause discrepancies during data handling [18].

5.3 Data Transmission

In UART, the mode of transmission is in the form of a packet [19].

The piece that connects the transmitter and receiver includes the creation of serial packets and controls those physical hardware lines [19].

A packet consists of a start bit, data frame, a parity bit, and stop bits [19].

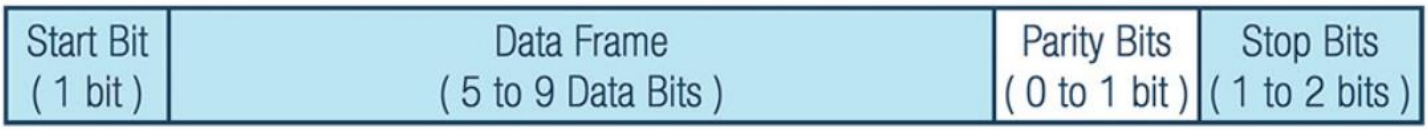


Figure 5.3 UART packet.

5.3.1 Start Bit

The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one (1) clock cycle [19].

When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate [19].

5.3.2 Data Frame

The data frame contains the actual data being transferred. It can be five (5) bits up to eight (8) bits long if a parity bit is used [19].

If no parity bit is used, the data frame can be nine (9) bits long. In most cases, the data is sent with the least significant bit first [19].

5.3.3 Parity

Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission [19].

Bits can be changed by electromagnetic radiation, mismatched baud rates, or long-distance data transfers [19].

After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number [19].

If the parity bit is a 0 (even parity), the 1 or logic-high bit in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bit or logic highs in the data frame should total to an odd number [19].

When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd, or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed [19].

5.3.4 Stop Bits

To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for one (1) to two (2) bit(s) duration [19].

5.4 Steps of UART Transmission

First: The transmitting UART receives data in parallel from the data bus.

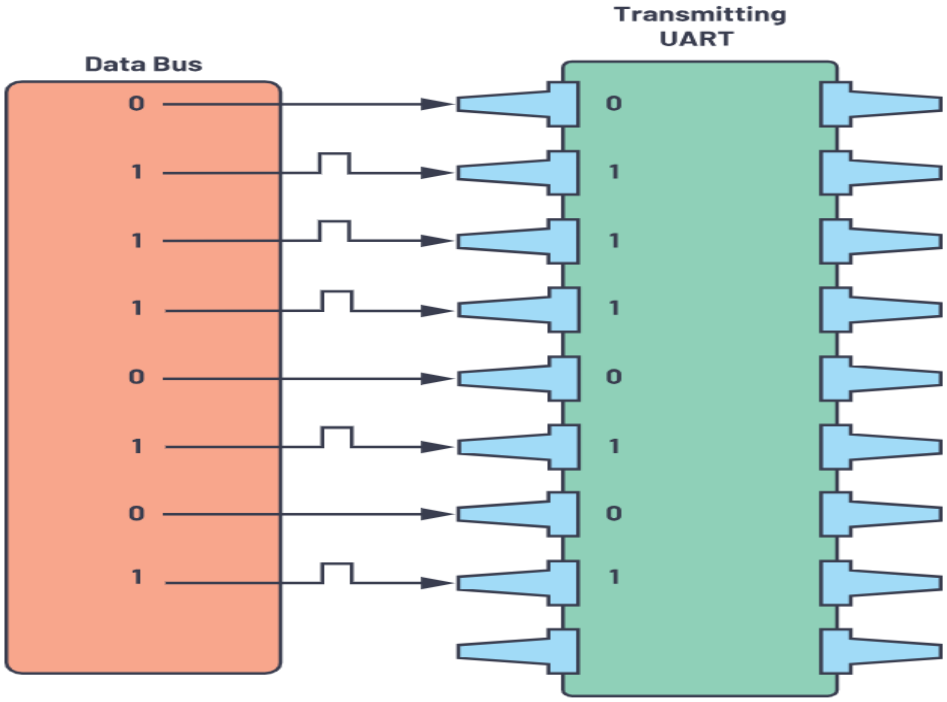


Figure 5.4 Data bus to the transmitting UART.

Second: The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame [19].

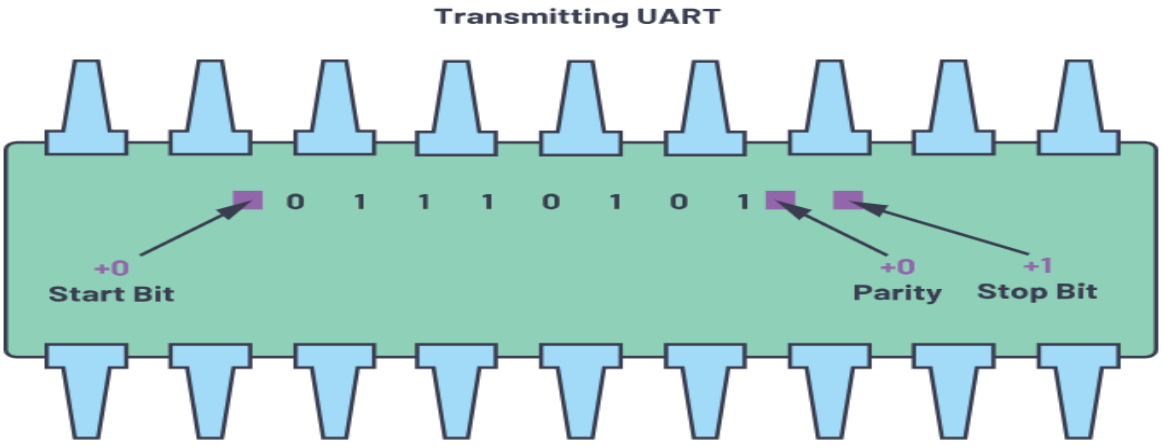


Figure 5.5 UART data frame at the Tx side.

Third: The entire packet is sent serially starting from start bit to stop bit from the transmitting UART to the receiving UART [19].

The receiving UART samples the data line at the preconfigured baud rate [19].

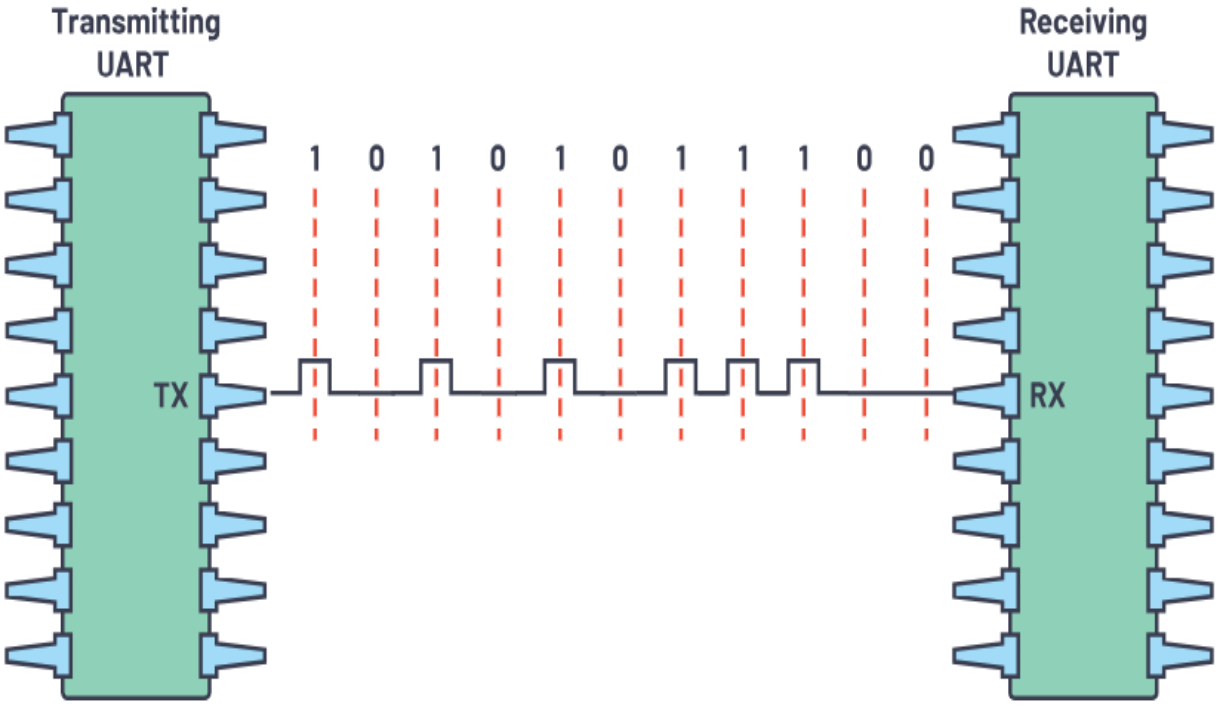


Figure 5.6 UART transmission.

Fourth: The receiving UART discards the start bit, parity bit, and stop bit from the data frame [19].

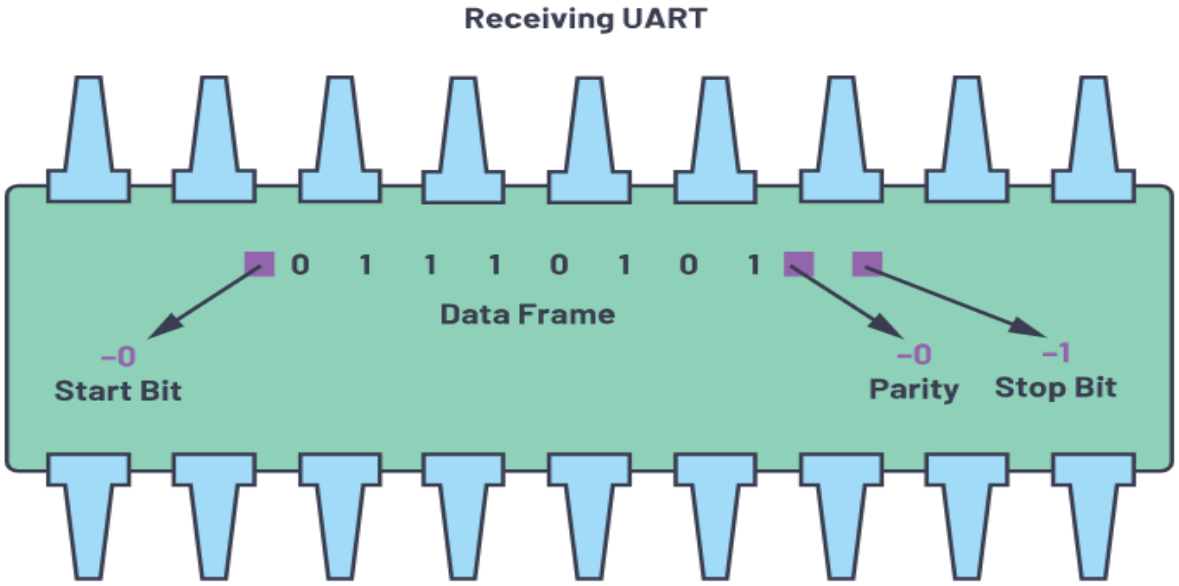


Figure 5.7 The UART data frame at the Rx side.

Fifth: The receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end [19].

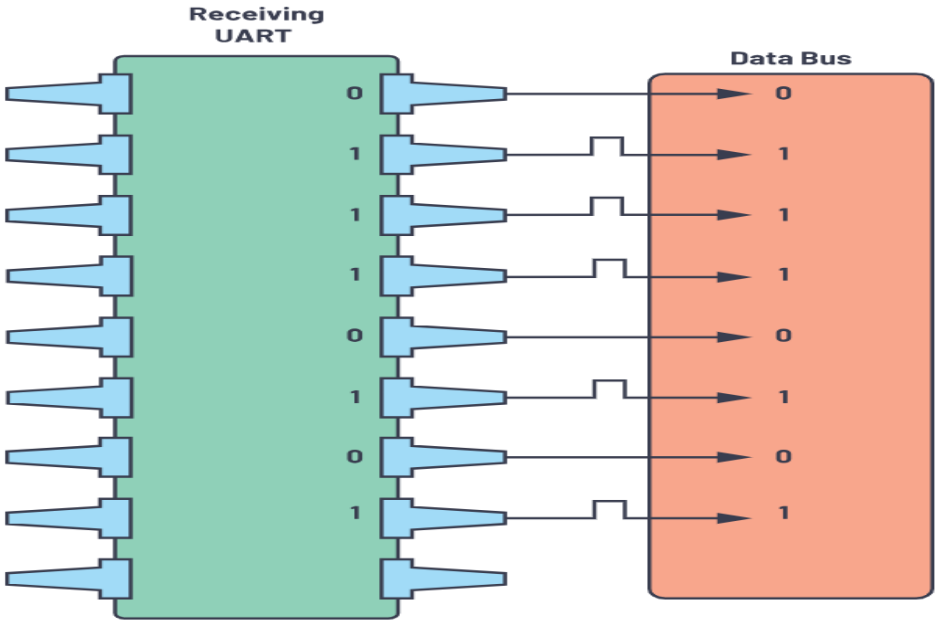


Figure 5.8 Receiving UART to data bus.

5.5 UART Operations

When using any hardware communication protocol, it's a prerequisite to check the data sheet and hardware reference manual [19].

Here are the steps to follow:

First: Check the data sheet interface of the device [19].

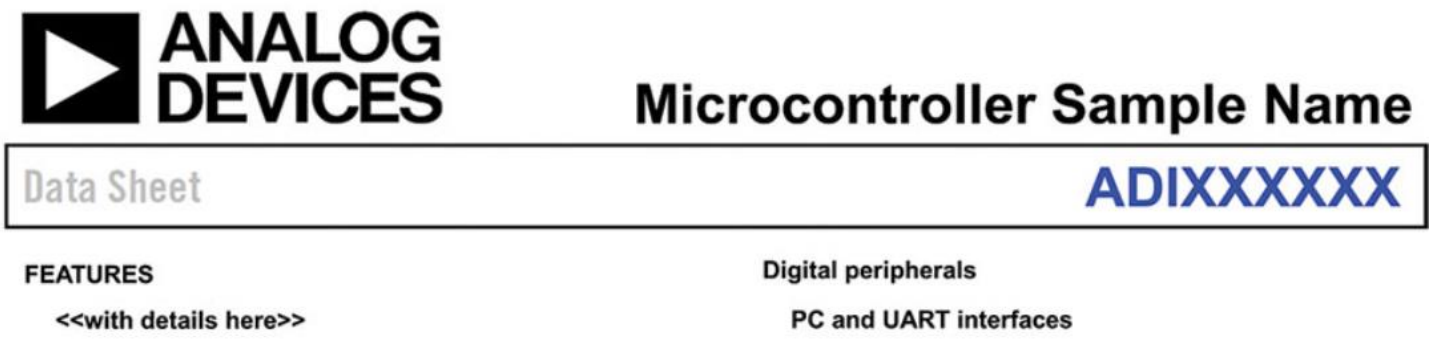


Figure 5.9 Microcontroller data sheet.

Second: Check the specific details for the UART PORT such as the operation mode, data bits length, the parity bit, and stop bits [19].

Sample UART port details in data sheet: UART Port [19].

The sample MCUs provide a full-duplex UART port, which is fully compatible with PC standard UARTs [19].

The UART port provides a simplified UART interface to other peripherals or hosts, supporting full-duplex, DMA, and asynchronous transfer of serial data [19].

The UART port includes support for five to eight data bits, and none, even, or odd parity [19].

A frame is terminated by one and a half or two stop bits [19].

Third: Check the UART operation details, including the baud rate computation. Baud rate is configured using the following sample formula [19].

This formula varies depending on the microcontroller [19].

Sample details of UART operations:

- 5 to 8 data bits
- 1, 2, or 1 and ½ stop bits
- None, or even or odd parity
- Programmable oversample rate by 4, 8, 16, 32
- Baud rate = PCLK/ ((M + N/2048) × $2^{OSR + 2}$ × DIV

Where:

OSR (oversample rate)

UART_LCR2.OSR = 0 to 3

DIV (baud rate divider)

UART_DIV = 1 to 65535

M (DIVM fractional baud rate M)

UART_FBR.DIVM = 1 to 3

N (DIVM fractional baud rate M)

UART_FBR.DIVN = 0 to 2047

Fourth: For the baud rate, make sure to check what peripheral clock (PCLK) to use. In this example, there is a 26 MHz PCLK and 16 MHz PCLK available [19].

Notice that OSR, DIV, DIVM, and DIVN varies per device [19].

Baud Rate	OSR	DIV	DIVM	DIVN
9600	3	24	3	1078
115200	3	4	1	1563

Table 5.2 Baud Rate Example Based on 26 MHz PCLK

Baud Rate	OSR	DIV	DIVM	DIVN
9600	3	17	3	1078
115200	3	2	2	348

Table 5.3 Baud Rate Example Based on 16 MHz PCLK

Fifth: Next part is to check the detailed registers for UART Configuration. Take a look at the parameters in computing the baud rate such as UART_LCR2, UART_DIV, and UART_FBR. Table 4 will lead to a specific register to cover [19].

Name	Description
UART_DIV	Baud rate divider
UART_FIBR	Fractional baud rate
UART_LCR2	Second line control

Table 5.4 UART Register Descriptions

Sixth: Under each register, check the details and substitute the values to compute for the baud rate, then start implementing the UART [19].

5.6 Advantages

- Only uses two wires [20].
- No clock signal is necessary [20].
- Has a parity bit to allow for error checking [20].
- The structure of the data packet can be changed as long as both sides are set up for it [20].
- Well documented and widely used method [20].

5.7 Disadvantages

- The size of the data frame is limited to a maximum of 9 bits [20].
- Doesn't support multiple slave or multiple master systems [20].
- The baud rates of each UART must be within 10% of each [20].

5.8 Use Cases

You can use UART for many applications, such as:

- Debugging: Early detection of system bugs is important during development. Adding UART can help in this scenario by capturing messages from the system [20].
- Manufacturing function-level tracing: Logs are very important in manufacturing. They determine functionalities by alerting operators to what is happening on the manufacturing line [20].
- Customer or client updates: Software updates are highly important. Having complete, dynamic hardware with up-to-update capable software is important to having a complete system [20].

Testing/verification: Verifying products before they leave the manufacturing process helps deliver the best quality products possible to customers [20].

Bluetooth

6

In this chapter, we'll talk about Bluetooth in general.

Introduction, How Bluetooth Work Bluetooth Profiles, Bluetooth Versions, Bluetooth architecture in ESP32.

6.1 Introduction

Figure 6.1 Bluetooth Icon

Bluetooth is a wireless communication protocol used for short-range communication between devices [21].
It was developed in the 1990s by Ericsson and has since become a widely adopted standard for wireless communication [21].

Bluetooth technology uses radio waves to transmit data over short distances, typically up to 10 meters (33 feet) but can extend up to 100 meters (328 feet) with Bluetooth 5.0[21].
The protocol allows devices to communicate with each other without the need for cables or wires [21].

The Bluetooth protocol is divided into several layers, including a physical layer that handles the radio transmission of data, a link layer that establishes connections between devices, and an application layer that defines the data that is transmitted between devices [21].

Bluetooth uses a frequency-hopping spread spectrum technique to reduce interference from other wireless devices operating in the same frequency band [21].

This means that the signal hops between different frequencies in the 2.4 GHz band, which reduces the likelihood of interference from other devices, this protocol is used in a wide range of applications, including mobile phones, wireless headphones, smart speakers, gaming controllers' devices [21].

6.2 How Bluetooth Work

The Bluetooth protocol operates at 2.4GHz in the same unlicensed ISM frequency band where RF protocols like ZIGBEE and WIFI also exist [21].

There is a standardized set of rules and specifications that differentiates it from other protocols [21].
If you have a few hours to kill and want to learn every nook and cranny of Bluetooth, check out the published specifications, otherwise here's a quick overview of what makes Bluetooth special [21].

Masters, Slaves

Bluetooth networks (commonly referred to as PICONETS) use a master/slave model to control when and where devices can send data [21].

In this model, a single master device can be connected to up to seven different slave devices [21].
Any slave device in the PICONET can only be connected to a single master [21].

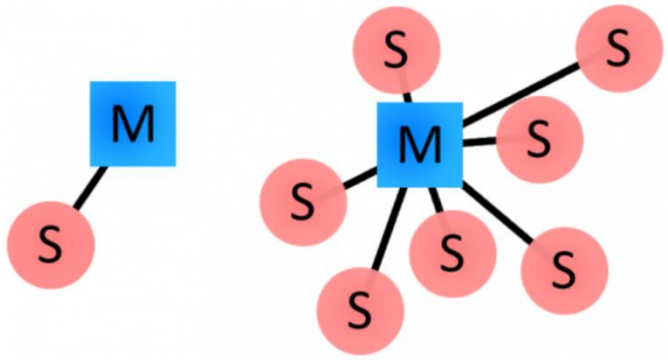


Figure 6.2 *Bluetooth master/slave topologies.*

The master coordinates communication throughout the PICONET [21].

It can send data to any of its slaves and request data from them as well [21].

Slaves are only allowed to transmit to and receive from their master [21].

They can't talk to other slaves in the PICONETS [21].

Bluetooth Addresses and Names

Every single Bluetooth device has a unique 48-bit address, commonly abbreviated BD_ADDR [21].

This will usually be presented in the form of a 12-digit hexadecimal value[21].

The most-significant half (24 bits) of the address is an organization unique identifier (OUI), which identifies the manufacturer [21].

The lower 24-bits are the more unique part of the address [21].

This address should be visible on most Bluetooth devices [21].

For example, on this RN-42 Bluetooth Module, the address printed next to "MAC NO." is 000666422152:

Figure 6.3 MAC NO.

The "000666" portion of that address is the OUI of Roving Networks, the manufacturer of the module [21].

Every RN module will share those upper 24-bits [21].

The "422152" portion of the module is the more unique ID of the device [21].

Bluetooth devices can also have user-friendly names given to them [21].

These are usually presented to the user, in place of the address, to help identify which device it is [21].

6.3 Bluetooth Profiles

Bluetooth profiles are additional protocols that build upon the basic Bluetooth standard to more clearly define what kind of data a Bluetooth module is transmitting [21].
While Bluetooth specifications define how the technology *works*, profiles define how it's *used* [21].
The profile(s) a Bluetooth device supports determine(s) what application it's geared towards [21].

A hands-free Bluetooth headset, for example, would use headset profile (HSP), while a Nintendo Wii Controller would implement the human interface device (HID) profile [21].

For two Bluetooth devices to be compatible, they **must support the same profiles** [21].

Let's take a look at a few of the more commonly encountered Bluetooth profiles.

Serial Port Profile (SPP)

If you're replacing a serial communication interface (like RS-232 or a UART) with Bluetooth, SPP is the profile for you[21].

SPP is great for sending bursts of data between two devices [21].

It’s one of the more fundamental Bluetooth profiles (Bluetooth's original purpose was to replace RS-232 cables after all) [21].

Using SPP, each connected device can send and receive data just as if there were RX and TX lines connected between them [21].

For example, could converse with each other from across rooms, instead of from across the desk [21].

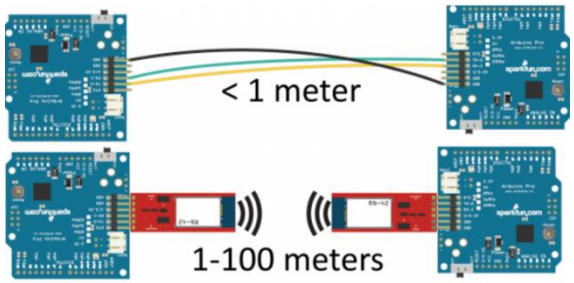


Figure 6.4 Serial Port Profile.

Human Interface Device (HID)

HID is the go-to profile for Bluetooth-enabled user-input devices like mice, keyboards, and joysticks [21].

It's also used for a lot of modern video game controllers, like PS3 controllers [21].

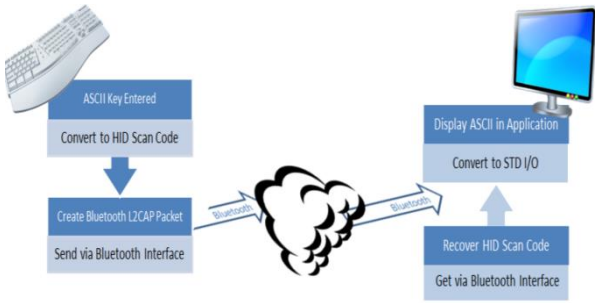


Figure 6.5 HID interface, from RN-42-HID User's Guide.

Bluetooth's HID profile is a riff on the HID profile already defined for human input USB devices [21].

Just as SPP serves as a replacement for RS-232 cables, HID aims to replace USB cables (a much taller task!) [21].

- **Hands-Free Profile (HFP) and Headset Profile (HSP)**

Those Bluetooth earpieces that make important business guys look like self-conversing wackos? Those usually use headset profile (HSP) or hands-free profile (HFP). HFP is used in the hands-free audio systems built into cars [21].

It implements a few features on top of those in HSP to allow for common phone interactions (accepting/rejecting calls, hanging up, etc.) to occur while the phone remains in your pocket [21].

- **Advanced Audio Distribution Profile (A2DP)**

Advanced audio distribution profile (A2DP) defines how audio can be transmitted from one Bluetooth device to another [21], where HFP and HSP send audio to and from both devices, A2DP is a one-way street, but the audio quality has the potential to be *much* higher [21].

A2DP is well-suited to wireless audio transmissions between an MP3 player and a Bluetooth-enabled stereo.

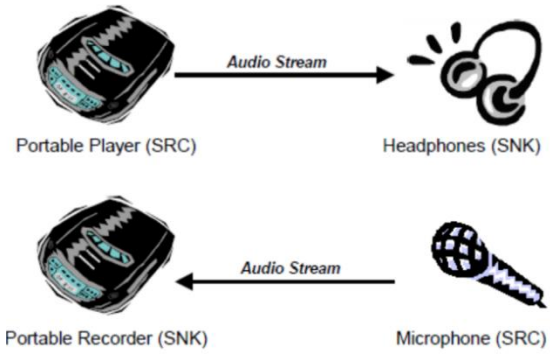


Figure 6.6 Image from A2DP specification (v1.3).

In the least they'll support SBC (sub band codec), they may also support MPEG-1, MPEG-2, AAC, and ATRAC [21].

A/V Remote Control Profile (AVRCP)

The audio/video remote control profile (AVRCP) allows for remote controlling of a Bluetooth device [21].

It's usually implemented alongside A2DP to allow the remote speaker to tell the audio-sending device to fast-forward, rewind, etc [21].

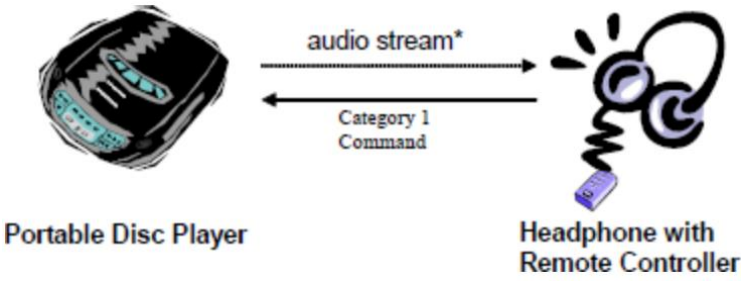


Figure 6.7 Image from AVRCP specification (v1.5).

6.4 Bluetooth Versions

Bluetooth has been constantly evolving since it was conceived in 1994[22].

The most recent update of Bluetooth, Bluetooth v4.0, is just beginning to gain traction in the consumer electronics industry, but some of the previous versions are still widely used [22].

Here's a rundown of the commonly encountered Bluetooth versions:

Bluetooth v1.2
The v1.x releases laid the groundwork for the protocols and specifications future versions would build upon. Bluetooth v1.2 was the latest and most stable 1.x version [22].

These modules are rather limited compared to later versions [22].

They support data rates of up to 1 Mbps (more like 0.7 Mbps in practice) and 10-meter maximum range [22].

Bluetooth v2.1 + EDR
The 2.x versions of Bluetooth introduced **enhanced data rate (EDR)**, which increased the data rate potential up to 3 Mbps (closer to 2.1 Mbps in practice) [22].

Bluetooth v2.1, released in 2007, introduced **secure simple pairing (SSP)**, which overhauled the pairing process [22].

Bluetooth v2.1 modules are still very common.

For low-speed microcontrollers, where 2 Mbps is still *fast*, v2.1 gives them just about everything they could need. The RN-42 Bluetooth module, for example, remains popular in products like the Bluetooth Mate[22].

Bluetooth v3.0 + HS
You thought 3 Mbps was fast? Multiply that by eight and you have Bluetooth v3.0's optimum speed -- 24 Mbps [22].

That speed can be a little deceiving though, because the data is actually transmitted over a WIFI (802.11) connection. Bluetooth is only used to establish and manage a connection [22].

It can be tricky to nail down the maximum data rate of a v3.0 device [22].

Some devices can be "Bluetooth v3.0+HS", and others might be labeled "Bluetooth v3.0"[22].

Only those devices with the "+HS" suffix are capable of routing data through WIFI and achieving that 24 Mbps speed [22].

"Bluetooth v3.0" devices are still limited to a maximum of 3 Mbps, but they do support other features introduced by the 3.0 standard like better power control and a streaming mode [22].

Bluetooth v4.0 and Bluetooth Low Energy
Bluetooth 4.0 split the Bluetooth specification into three categories: classic, high-speed, and low energy [22].

Classic and high-speed call back to Bluetooth versions v2.1+EDR and v3.0+HS respectively [22].

The real standout of Bluetooth v4.0 is **Bluetooth low energy (BLE)** [22].

BLE is a massive overhaul of the Bluetooth specifications, aimed at very low power applications [22].

It sacrifices range (50m instead of 100m) and data throughput (0.27 Mbps instead of 0.7-2.1 Mbps) for a significant savings in power consumption [22].

BLE is aimed at peripheral devices which operate on batteries, and don't require high data rates, or constant data transmission [22].

Smart watches, like the Meta Watch , are a good example of this application [22].

6.5 Bluetooth architecture in ESP32

6.5.1 Bluetooth Application Structure

Bluetooth is a wireless technology standard for exchanging data over short distances, with advantages including robustness, low power consumption and low cost [22].

The Bluetooth system can be divided into two different categories: Classic Bluetooth and Bluetooth Low Energy (BLE). ESP32 supports dual-mode Bluetooth, meaning that both Classic Bluetooth and BLE are supported by ESP32[22].

Basically, the Bluetooth protocol stack is split into two parts: a “controller stack” and a “host stack”.

The controller stack contains the PHY, Baseband, Link Controller, Link Manager, Device Manager, HCI and other modules, and is used for the hardware interface management and link management.

The host stack contains L2CAP, SMP, SDP, ATT, GATT, GAP and various profiles, and functions as an interface to the application layer, thus facilitating the application layer to access the Bluetooth system [22].

The Bluetooth Host can be implemented on the same device as the Controller, or on different devices [22].

NOTE: Both approaches are supported by ESP32[22].

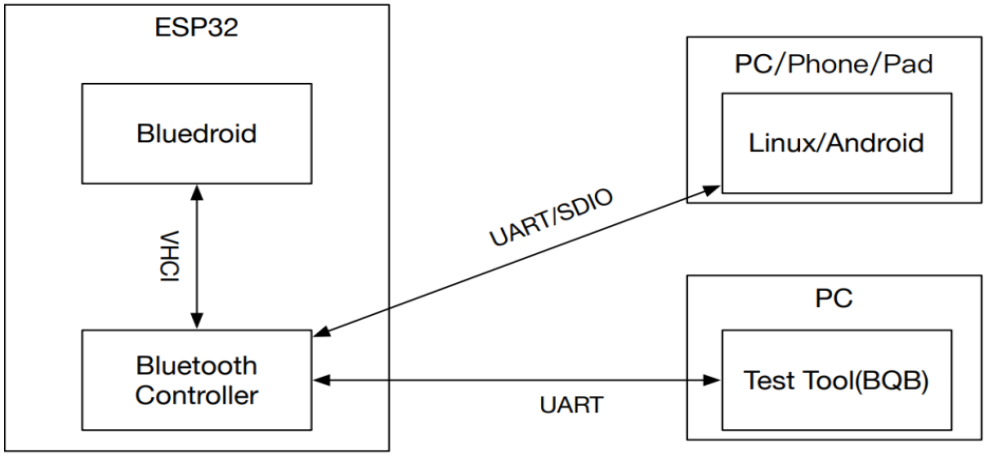


Figure 6.8 the architecture of Bluetooth host and controller in ESP-IDP.

• **Scenario 1 (Default ESP-IDF setting):** BLUEDROID is selected as the Bluetooth Host, and VHCI (software-implemented virtual HCI interface) is used for the communication between Bluetooth Host and Controller [22].

In this scenario, both the BLUEDROID and the Controller are implemented on the same device (i.e., the ESP32 chip), eliminating the need for an extra PC or other host devices running the Bluetooth Host [22].

• **Scenario 2:** the ESP32 system is used only as a Bluetooth Controller, and an extra device running the Bluetooth Host is required (such as a Linux PC running Blue Z or an Android device running BLUEDROID) [22].

In this scenario, Controller and Host are implemented on different devices, which is quite like the case of mobile phones, PADs, or PCs [22].

• **Scenario 3:** This scenario is like Scenario 2[22].

The difference lies in that, in the BQB controller test (or other certifications), ESP32 can be tested by connecting it to the test tools, with the UART being enabled as the IO interface [22].

6.5.2 Selection of the HCI Interfaces

In the ESP32 system, only one IO interface at a time can be used by HCI, meaning that if UART is enabled, other interfaces such as the VHCI and SDIO are disabled [22].

In the ESP-IDF (V2.1), you can configure the Bluetooth HCI IO interface as VHCI or UART in the menu screen, as shown below:

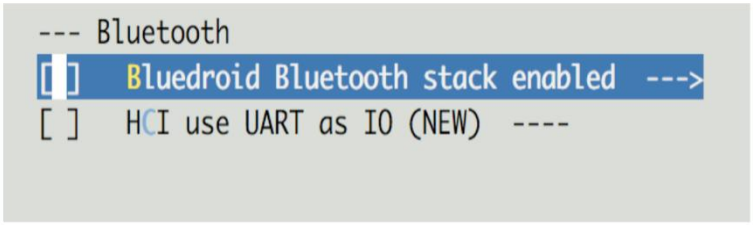


Figure 6.9 Configuration of the HCI IO interface.

When the Blue droid Bluetooth stack enabled option is selected, VHCI is enabled as the IO interface and the HCI use UART as IO (NEW) option will disappear from the menu [22].
When the HCI uses UART as IO (NEW) option is selected, UART is enabled as the IO interface [22].

Currently, other IOs are not supported in ESP-IDF [22].

If you want to use other IOs, such as the SPI, a SPI-VHCI bridge is required.

Option 1:
When the Blue droid Bluetooth stack enabled option is selected, the following screen is displayed:

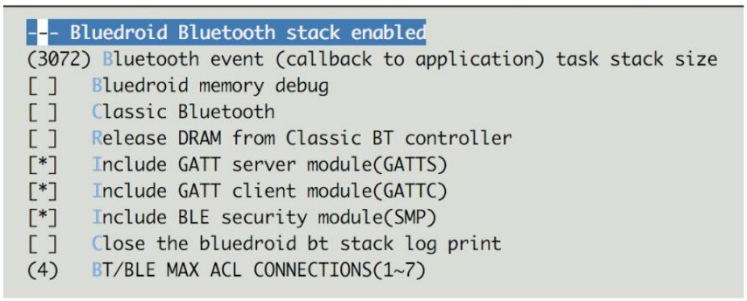

-*- Bluedroid Bluetooth stack enabled
(3072) Bluetooth event (callback to application) task stack size
[ ] Bluedroid memory debug
[ ] Classic Bluetooth
[ ] Release DRAM from Classic BT controller
[*] Include GATT server module(GATTS)
[*] Include GATT client module(GATTC)
[*] Include BLE security module(SMP)
[ ] Close the bluedroid bt stack log print
(4) BT/BLE MAX ACL CONNECTIONS(1~7)

Figure 6.10 VHCI configuration.

Here, users can configure the following items:

- Bluetooth event (callback to application) task stack size sets the size of the BTC Task).
- Bluedroid memory debug debugs the BLUEDROID memory.
- Classic Bluetooth: enables Classic Bluetooth [22].
- Release DRAM from Classic BT Controller: releases the DRAM from the Classic Bluetooth Controller [22].
- Include GATT server module (GATTS): includes the GATTS module.
- Include GATT client module (GATTC): includes the GATTC module.
- Include BLE security module (SMP): includes the SMP module.
- Close the blue droid stack log print closes the BLUEDROID printing.
- BT/BLE MAX ACL CONNECTIONS (1~7): sets the maximum number of ACL connections [22].

Option 2:
When the HCI use UART as IO option is selected, the following screen is displayed:

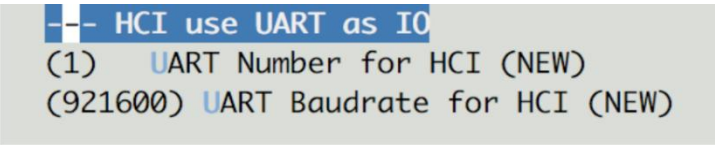


Figure 6.11 UART configuration.

Users can configure the "UART Number for HCI (NEW)" (UART port number) and the "UART Baud rate for HCI (NEW)" (the baud rate of UART) here.

It should also be mentioned here that CTS/RTS must be supported, to enable the UART as the HCI IO interface [22].

6.5.3 Bluetooth Operating Environment

The default operating environment of ESP-IDF is dual-core Free RTOS.

ESP32 Bluetooth can assign function-based tasks with different priorities. The tasks with the highest priority are the ones running the Controller.

The Controller tasks, which have higher requirements of real time, have the highest priority in the Free RTOS system except for the IPC tasks, which are mainly for the inter process communication between the dual-core CPUs.

BLUEDROID (the default ESP-IDF Bluetooth Host) contains four tasks in total, which run the BTC, BTU, UPWARD, and HCI DOWNWARD [22].

Communication between ESP32 and Raspberry Pi

7

In this chapter, we'll talk about the communication between the two ESP32 and Which communication mode is the best, and how Raspberry Pi communicates or take the data from ESP32 in each car.

7.1 The Communication Between The 2 ESP32

7.1.1 ESP32 in Our System

In our system we have two cars, each of them can be (the ego car) or (the remote Car), and each car contains one ESP32 (The module we use to make the two cars communicate with each other).
ESP32 communicates with each other using ESP NOW communication protocol that makes a private network without WIFI or a Router.

7.1.2 ESP NOW

ESP-NOW is a kind of connectionless Wi-Fi communication protocol that is defined by Espressif. Different from traditional Wi-Fi protocols, the first five upper layers in OSI are simplified to one layer in ESP-NOW [17].

So, the data does not need to go through the physical layer, data link layer, network layer, transport layer in turn, which reduces the delay caused by packet loss under congested network and leads to fast response time [17].

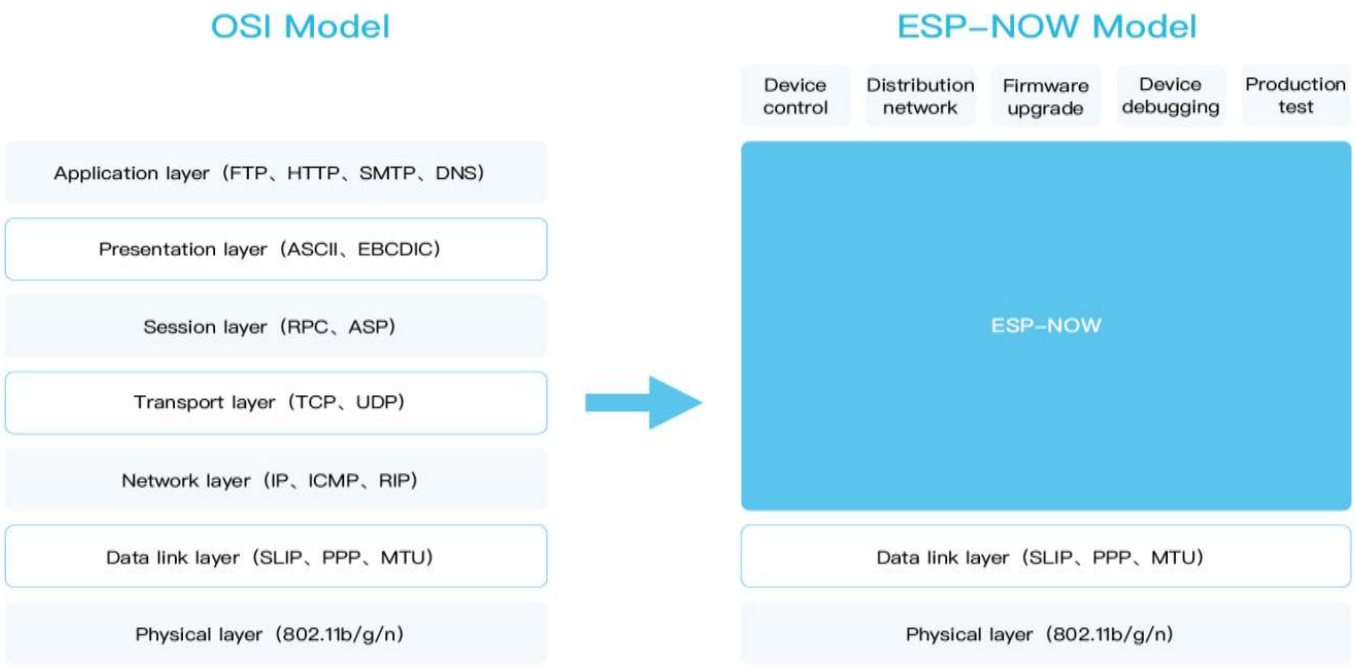


Figure 7.1 OSI Model And ESP-NOW Model

ESP-NOW occupies less CPU and flash resource. It can work with Wi-Fi and Bluetooth LE, and supports the series of ESP8266、ESP32、ESP32-S and ESP32-C [17].

The data transmission mode of ESP-NOW is flexible including unicast and broadcast and supports one-to-many and many-to-many device connection and control [17].

ESP-NOW can also be used as an independent auxiliary module to help network configuration, debugging and firmware upgrades [17].

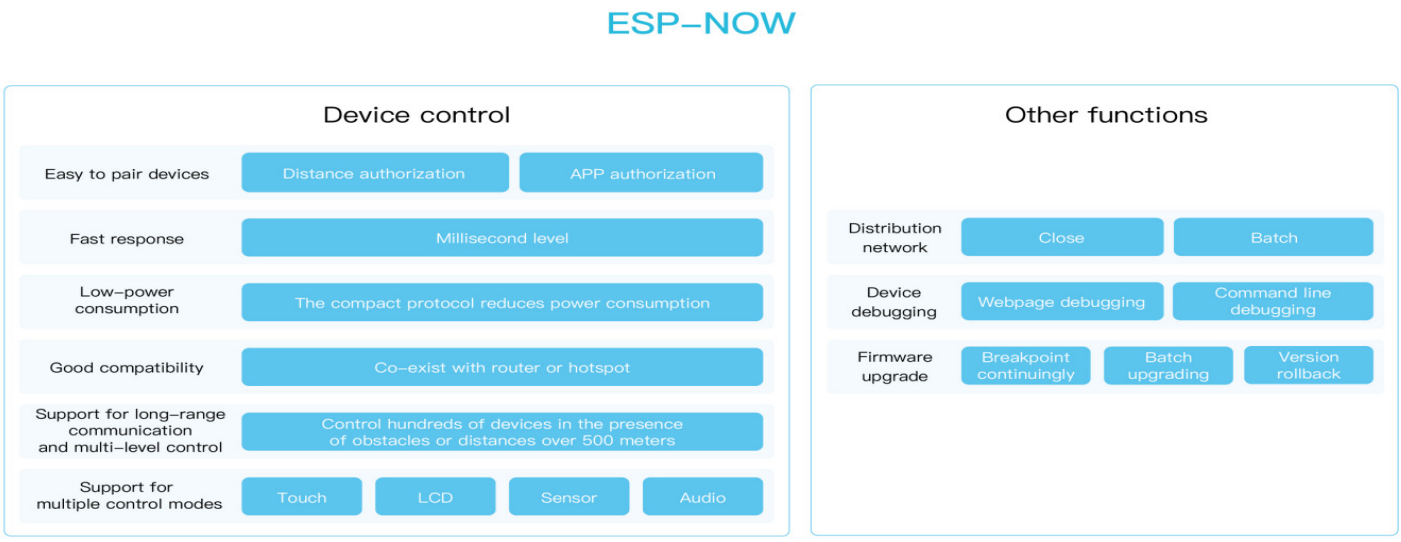


Figure 7.2 ESP-NOW Functions

7.1.3 Broadcast Communication Mode

As ESP32 uses ESP-NOW to communicate, and ESP-NOW uses Mac Address to achieve this communication. Thus, ESP32s need to know the Mac Addresses.
We can achieve this using three methods, but we use the best method that achieves our system requirements and real-life requirements also.

The first method:

In our system we have two cars only, so it`s easy to know the mac address of them. Then we can use the Two-Way Communication mode of ESP32, the bidirectional, or full-duplex, communication.

But as practical and real-life thinking, there will be more than 2 cars, a lot of cars.

The second method:

To solve the problem of the first method we will use another mode of communication which is Two-Way Networking.
It`s like a Two-Way Communication mode but between more boards, so now achieves the real-life requirements.

Again it`s also hard to know the mac address of each ESP32 (car) in real life, so that we will not use this method.

The third method:

To solve the problem of the second method we will use another mode of communication which is Broadcast Communication Mode.
It`s a fashion of One Initiator & Multiple Responders mode, The Initiator initiates a broadcast message to communicate with all the Responders.

Our system:

This method will achieve our system requirements, the ego car can communicate with remote car even without knowing the mac address of the remote car.
As we are using broadcast mode, we don't need to know the other boards' MAC addresses, so every board can run identical code.
We replace the mac address of the responders with constant address, expresses on the broadcast message.

Real life:

This method will achieve the real-life requirements Easley without any hardness, as we don`t need the mac address know.

7.2 The Communication Between the Raspberry Pi and the ESP32

Raspberry Pi communicates with ESP32 using UART (Universal Asynchronous Receiver/Transmitter) communication protocol.

7.2.1 UART in ESP32

ESP32 has three UART interfaces, i.e., UART0, UART1, and UART2, which provide asynchronous communication (RS232 and RS485) and IrDA support, communicating at a speed of up to 5 Mbps [17].

UART provides hardware management of the CTS and RTS signals and software flow control (XON and XOFF) [17].

Each UART controller is independently configurable with parameters such as baud rate, data bit length, bit ordering, number of stop bits, parity bit, etc [17].
All the regular UART controllers are compatible with UART-enabled devices from various manufacturers and can also support Infrared Data Association (IrDA) protocol [17].

We are using the Arduino IDE to develop and burn the code of UART configuration of ESP32 in the two cars [17].

7.2.2 UART in Raspberry Pi

UART is commonly used on the Raspberry Pi as a convenient way to control it over the GPIO or access the kernel boot messages from the serial console (enabled by default) [23].
Raspberry Pi 2/3 has two UARTs, uart1 and uart0. Raspberry Pi 4 has four additional UARTs available. Only uart0/1 is enabled over GPIO pin 14/15 by default. Also, UARTs can be enabled through the device tree overlays [23].

We developed the device driver of UART to configure the UART in Raspberry Pi, and then insert it in the image.

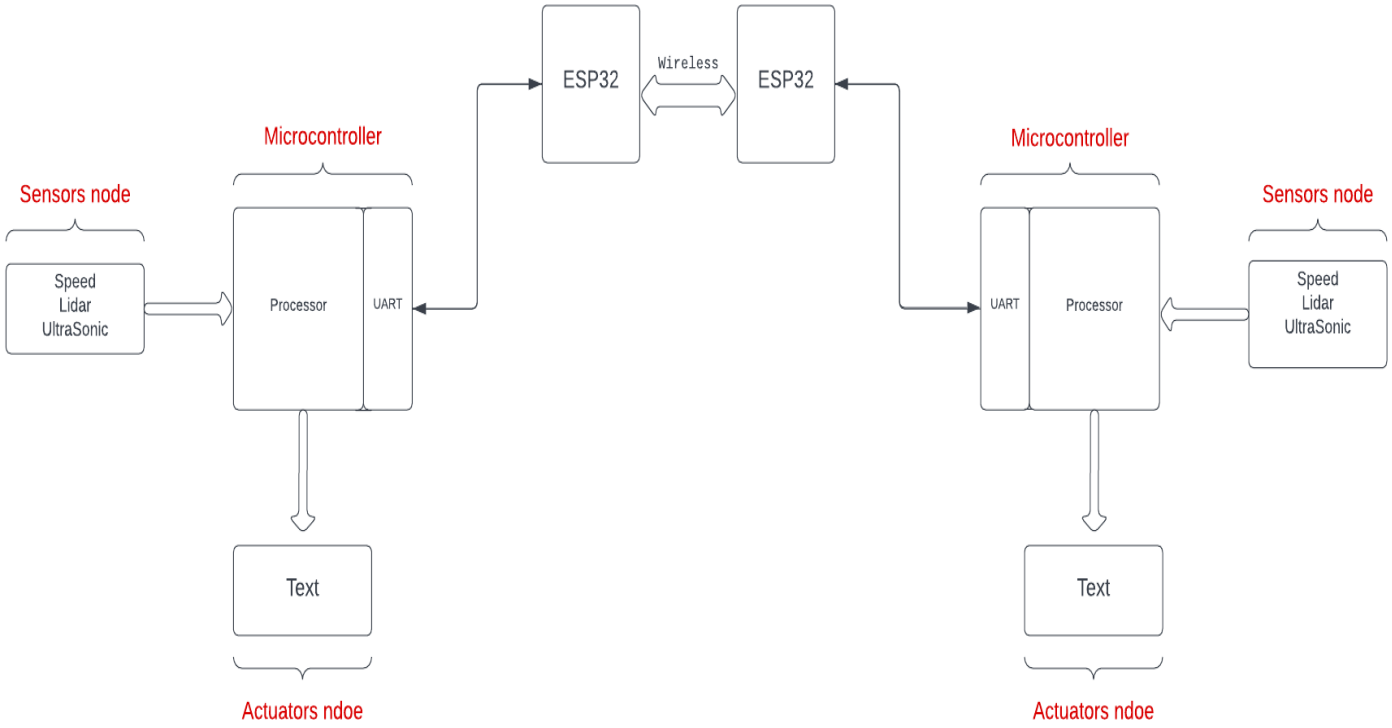


Figure 7.3 System layout

7.3 The Shape of The Message

In our system we have three messages, use them to define the current happening subsystem.
The message consists of two parts:

- **The first part**: is the direction of the ego car (the sender car).
- **The second part**: is the id of the current subsystem that causes the problem in the ego car.

We need the direction in three subsystems (EEBL, FCW, IMA), so we send it with the direction. We have three IDs:

i. ID (A): for subsystems EEBL and FCW.
ii. ID (B): for subsystems DNPW and BSW.
iii. ID (C): for subsystem IMA.

Message Format

Field	Number of Digits	Possible Data
Direction	**3**	**0, 90, 180 or 270**
ID	**1**	**A, B or C**

IC

SUB - SYSTEMS

CONTENT :

Chapter8 :EEBL & FCW
Chapter9 :BSW & DNPW
Chapter10:IMA

8 EEBL&FCW

In this Chapter we are going to talk about the first the second Real-life Scenarios we have implemented in our Project which are EEBL and FCW.

8.1 EEBL

8.1.1 Description of Alternatives

Every single day we always face the following situation, when there is a hard-braking vehicle in the pass ahead, or any object just came out of nowhere in front of another vehicle, the driver of this vehicle may not be able to get a proper chance to brake his car and avoid collisions that may occur, Here comes the use of our system “Emergency Electronic Brake”
When such a situation occurs, The Rear vehicle will automatically send a warning through an actuator to the driver when the distance between the two vehicles a threshold distance, If the driver did not react properly to this waring and both cars still getting closer, then for a certain distance the ECU will automatically brake the care without any control from the driver side [24].

Fig 8.1 EEBL

8.2 FCW

8.2.1 Description of Alternatives

Forward Collision Warning (FCW) is a safety feature that is commonly used in V2V Collision Avoidance systems. FCW uses sensors to detect the distance between the driver's vehicle and the vehicle in front of it. The system then analyzes the speed and position of both vehicles to determine if a collision is imminent [24].

If the system detects that a collision is likely to occur, it will issue a warning to the driver in the form of an audible and/or visual alert. The warning is designed to give the driver time to react and take evasive action, such as applying the brakes or changing lanes, to avoid the collision [24].

If both vehicles are too close, the warning signs will be triggered. When this occurs, the warning reads from “safe” to “critical,” which indicates an imminent collision [24].
FCW technology has been shown to be effective in reducing the number of rear-end collisions [24].

Fig8.2 FCW

8.3 EEBL&FCW

8.3.1 Description of Alternatives

This system is considered to be the most critical system in our project. For certain circumstances-especially for FCW- an urgent response must be taken automatically from the system itself without waiting for driver decisions [24].
In this system we simulate two real life scenarios. **First scenario** take place when a hard-braking vehicle in the path ahead in front of another vehicle and both vehicles are moving with the same direction, Hence the driver of rear vehicle may not be able to get a proper chance to use brakes and a collision may occur, Here comes the use of **"EEBL"** [24].
When such a situation occurs and because we are focusing on V2V Communication, when the distance between the two vehicles less than a threshold distance, The Rear vehicle will automatically send a warning to the driver in car in front that there is a car nearby in behind [24].
Regrading to the **Second scenario**, it takes place when two cars are on the same lane departure and heading to each other [24].
"FCW" will handle this scenario, by sending warnings to each other and forcing both cars to automatically brake if the distance between them is less than threshold distance [24].

Fig8.3 EEBL Scenario

Fig8.4 FCW Scenario

It's obvious that we can differentiate between EEBL and FCW by the direction of both cars and to implement this system we have two separated algorithms: Main algorithm and ID algorithm [24].

8.3.2 Main algorithm

In the main algorithm we are focused on whether my car is going to cause any possible EBBL or FCW Scenarios for any nearby car. If yes, this algorithm will be ended by sending the possible scenario's ID to the other car[24].

- **Flow Chart**

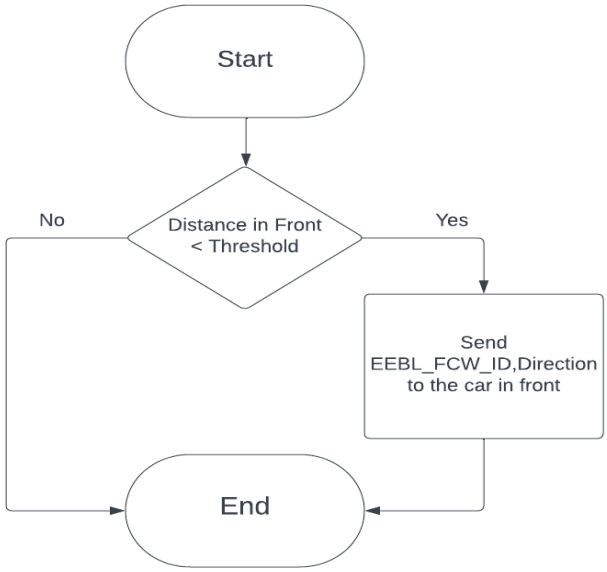


Fig.8.5 Flow Chart of Main Algorithm.

- **Pseudo Code**

```
BEGIN
READ   In-front Distance
IF   In-front Distance < Threshold Distance
CALL   SendID(EEBL_FCW_ID,Direction)
ELSE   /* DO NOTHING*/
ENDIF
END
```

Fig.8.6 Pseudo Code of Main Algorithm.

- **Block Diagram**

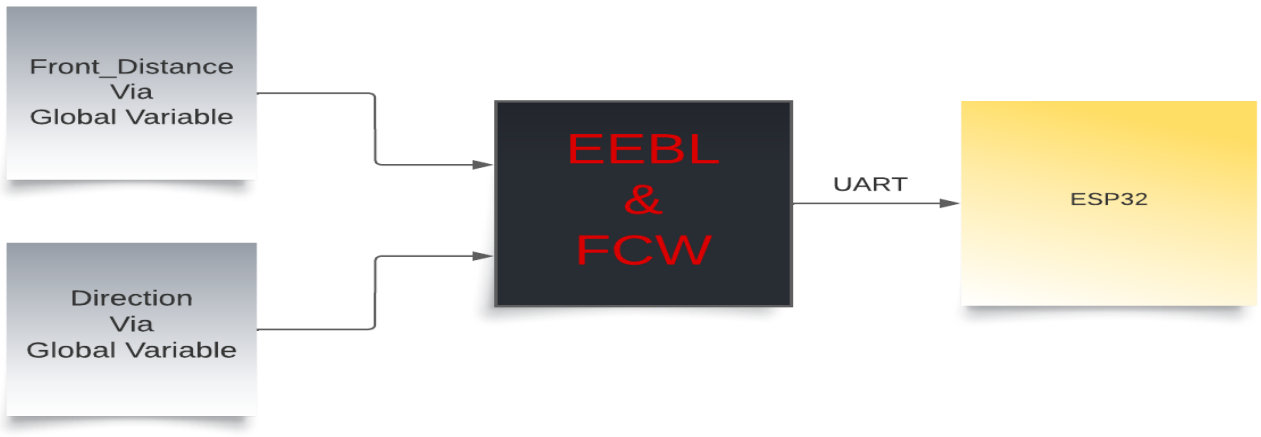


Fig.8.7 Block Diagram of Main Algorithm.

8.3.3 ID Algorithm

Secondly, we go into the Algorithm of the decided system. In The ID Algorithm we focus on the system itself, we already know the ID of the 2 systems (EEBL &FCW) so from the direction we decide which system is happening now [24].

- **Flow Chart**

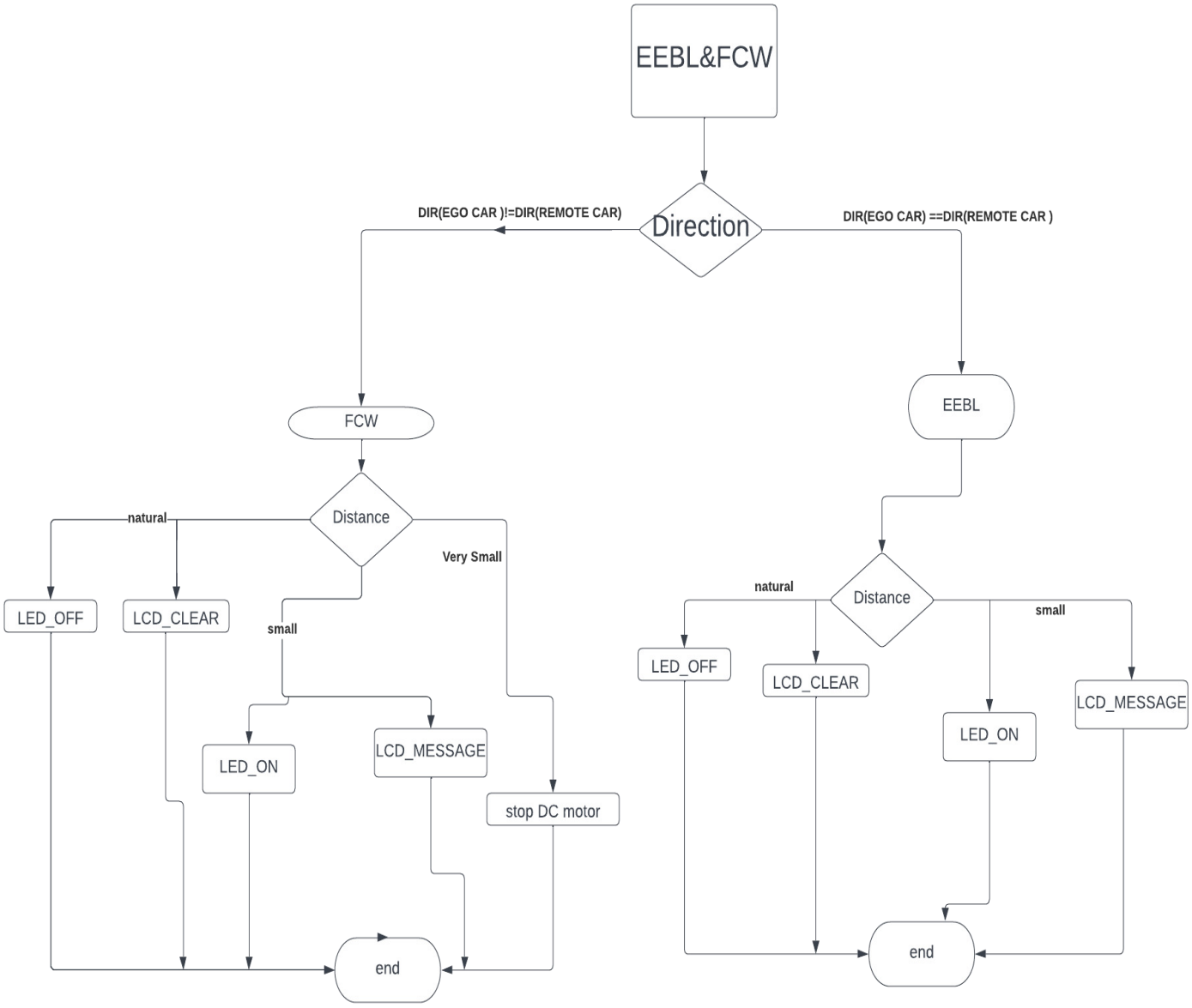


Fig8.8 Flow Chart of ID Algorithm.

- **Block Diagram**

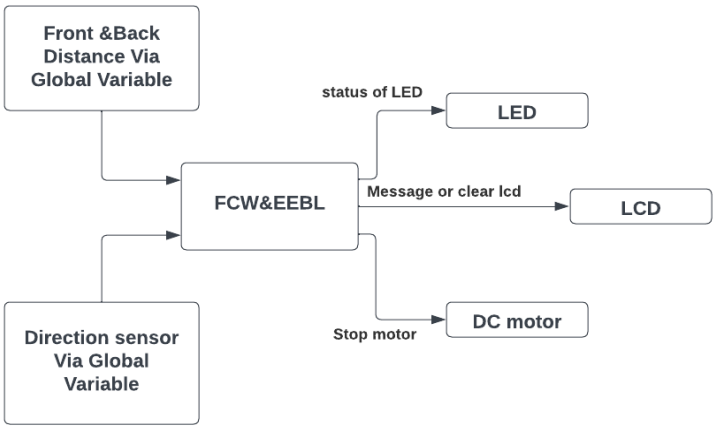


Fig.8.9 Block Diagram of ID algorithm

- **Pseudo Code**

```
BEGIN
READ      direction_remote
IF   direction _remote==direction_ego
CALL      EEBL_Do_Action();
ELSE
 CALL     FCW  Do  Action();
ENDIF
END
```

Fig.8.10 Pseudo Code of ID Algorithm.

8.3.4 EEBL&FCW Layered Architecture

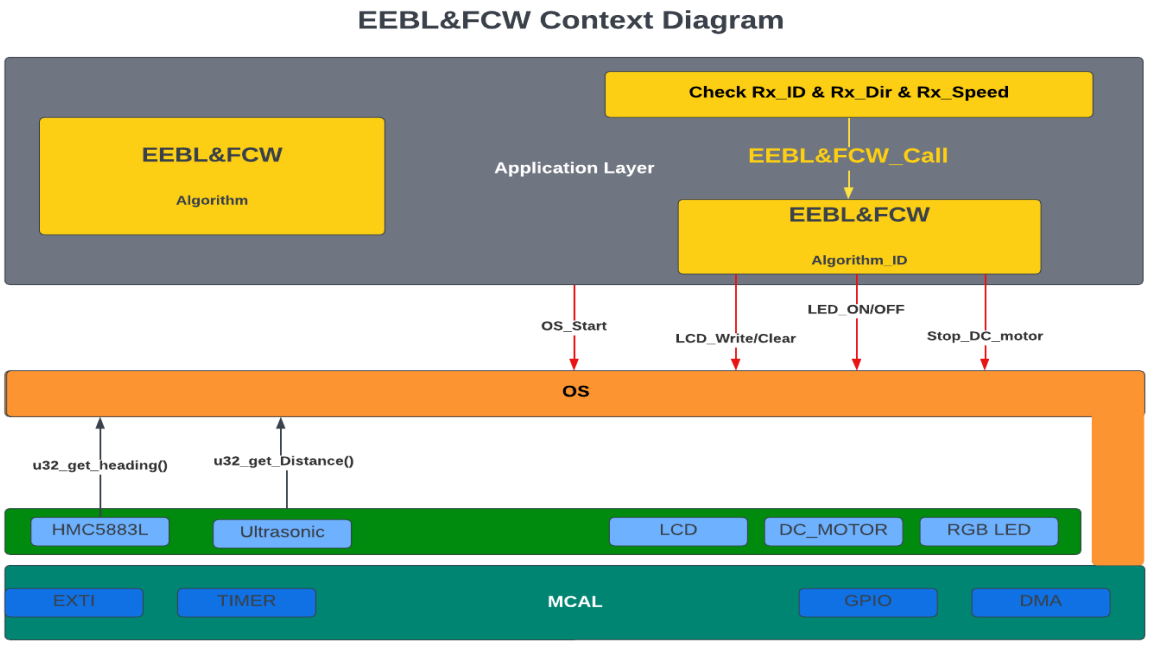


Fig.8.11 EEBL&FCW Context Diagram

8.3.5 Systems APIs Description

1. EEBL_FCW_vidTask

API Des.	responsible for checking if there is a possible EEBL or FCW Scenarios caused by my car.
API Input	Nothing.
API return	Nothing.

2. EEBL_FCW_vidAlgId

API Des.	responsible for checking the direction of both cars & decide if it's a FCW, EEBL and call it's Func.
API Input	Nothing.
API return	Nothing.

3. EEBL_vidAction

API Des.	responsible for checking the Rear_Ultrasonic distance and act accordingly.
API Input	Nothing.
API return	Nothing.

4. FCW_vidAction

API Des.	responsible for checking the Front_Ultrasonic distance and act accordingly.
API Input	Nothing.
API return	Nothing.

5. LCD_vidDisplayString

API Des.	responsible for display string.
API Input	Pointer to array of characters.
API return	Nothing.

6. RGB_vidSetSysColour

API Des.	responsible for blinking with the current subsystem predefined color.
API Input	Index of the current running sub-system.
API return	Nothing.

7. Stop_DC_Motor

API Des.	responsible for turning off Vehicle Motor.
API Input	Nothing.
API return	Nothing.

8. LCD_vidClearString

API Des.	responsible for clear LCD.
API Input	Nothing.
API return	Nothing.

9. RGB_vidTurn_Off

API Des.	responsible for Turn off LED
API Input	Index of the current running sub-system.
API return	Nothing.

BSW-DNPW

9

In this Chapter we are going to talk about the third the fourth Real-life Scenarios we have implemented in our Project which are BSW and DNPW.

9.1 BSW-DNPW

9.1.1 System Description

In this system we simulate **two real life scenarios** [24].

First scenario take place when there are areas around the car that are not visible to the driver through the side or rearview mirrors, Hence the driver of front vehicle may not be able to see a back vehicle and a collision may occur, Here comes the use of "BSW" [24]. Fig9.1

When such a situation occurs and because we are focusing on V2V Communication, when the distance between the two vehicles less than a threshold distance, The Rear vehicle will automatically send a warning to the driver in car in front that there is a car is in the blind spot [24].

Regarding to the **Second scenario**, take place when a vehicle in the path in front of another vehicle and both vehicles are moving with the same direction, hence systems are designed to detect and alert drivers who attempt to pass other vehicles in areas where passing is unsafe or prohibited.

Here comes the use of "DNPW" [24]. Fig9.2

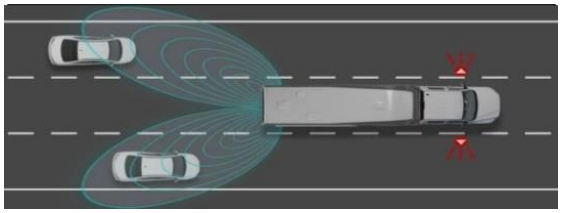


Figure9.1 BSW

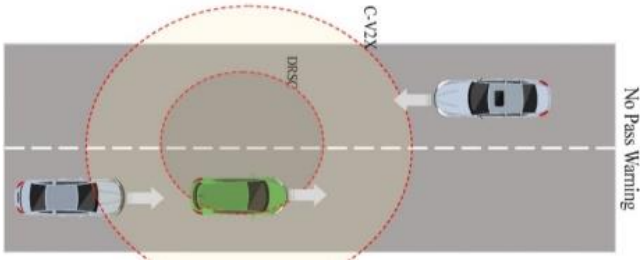


Figure 9.2 DNPW

It's obvious that we can differentiate between BSW and DNPW by the back sensors of both cars and to implement this system we have two separated algorithms: Main algorithm and ID algorithm [24].

9.1.2 Main algorithm

In the main algorithm we are focused on whether my car is going to cause any possible BSW or DNPW Scenarios for any nearby car [24].

If yes, this algorithm will be ended by sending the possible scenario's ID to the other car[24].

- **Flow Chart**

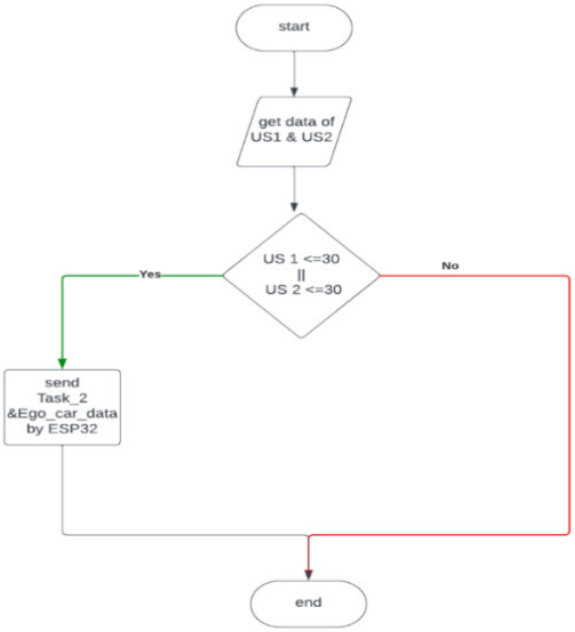


Figure 9.3 Flow Chart Of Main Algorithm.

- **Pseudo Code**

```
BEGIN
READ  In-front Distance
IF    ( (Right-front Distance < Threshold Distance) or
        (Left-front  Distance < Threshold Distance) )
CALL  SendID(BSW_DNPW_ID,Direction)
ELSE  /* DO NOTHING*/
ENDIF
END
```

Figure 9.4 Pseudo Code of Main Algorithm

- **Block Diagram**

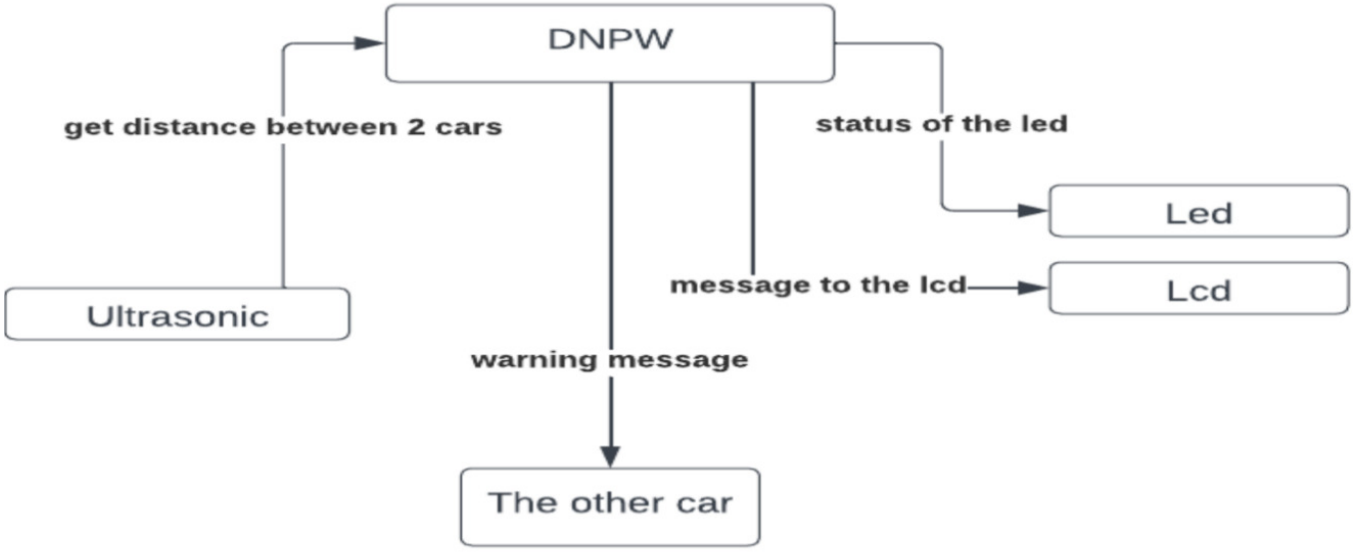


Figure 9.5 Block Diagram of Main Algorithm.

9.1.3 ID Algorithm

In The ID Algorithm we focus on the system itself, we already know the ID of the 2 systems (BSW &DNPW) so from Back Sensors we decide which system is happening now[24].

- Flow Chart

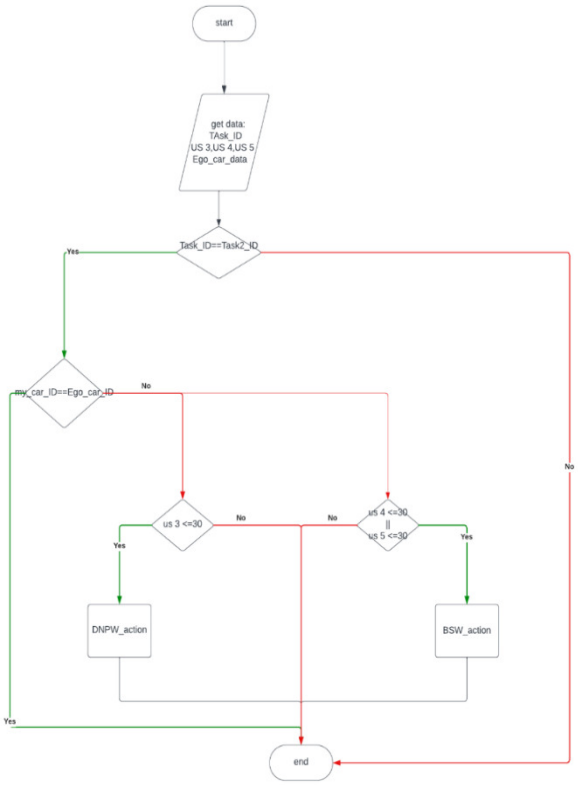


Figure 9.6 Flow Chart of ID Algorithm

- Pseudo Code

```
BEGIN
READ   In-front Distance
IF  Task_ID == Task2_ID
  IF  My_Car_ID == Ego_Car_ID
    /* DO NOTHING*/
  ELSE
    IF  Front_US Dis < Threshold_Dis
        CALL    DNPW_Do_Action();
    IF  Left_Back_US Dis < Threshold_Dis
        CALL    BSW_Do_Action();
ELSE   /* DO NOTHING*/
ENDIF
END
```

Figure 9.7 Pseudo Code of ID Algorithm

9.2 Layered Architecture

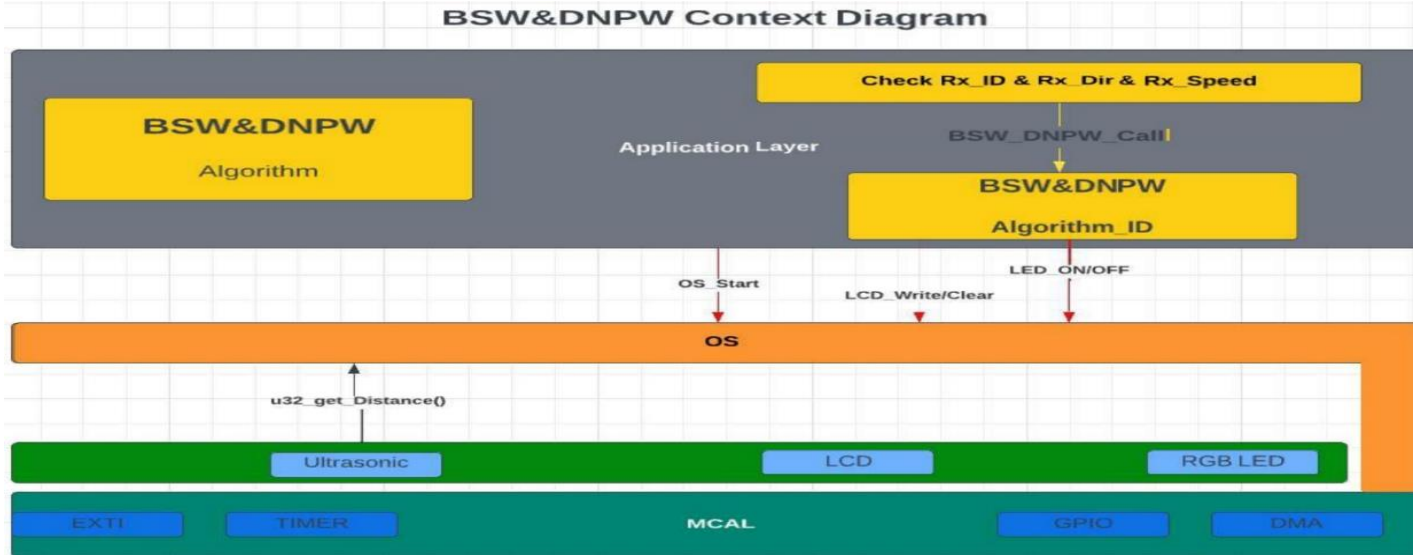


Figure 9.8 the BSW-DNPW Layered Architecture.

9.3 APIs Description

9.4

1. BSW-DNP _ vidTask

API Des	Responsible for checking if there is a possible BSW or DNPW Scenarios caused by my car.
API Input	Nothing.
API return	Nothing.

2. BSW-DNPW _vidAlgId

API Des	Responsible for checking the direction of both cars & decide if it's a BSW&DNPW and call it's Function.
API Input	Nothing.
API return	Nothing.

3. DNPW _vidAction

API Des	Responsible for checking the Front Ultrasonic distance and act accordingly.
API Input	Nothing.
API return	Nothing.

4. BSW_vidAction

API Des	Responsible for checking the Back_ Left & Right _Ultrasonic distance and act accordingly.
API Input	Nothing.
API return	Nothing.

5. LCD_ vidDisplayString

API Des	Responsible for display string.
API Input	Pointer to array of characters.
API return	Nothing.

6. LCD_ vidClearString

API Des	Responsible for clear LCD.
API Input	Nothing.
API return	Nothing.

7. RGB_ vidSetSysColour

API Des	Responsible for blinking with the current subsystem predefined color.
API Input	Index of the current running sub-system.
API return	Nothing.

8. RGB_ vidTurn_ Off

API Des	Responsible for Turn off LED.
API Input	Nothing.
API return	Nothing.

IMA

10

In the Chapter we are going to talk about the last scenario implemented in our project which is Intersection Movement Assist.

10.1 IMA

10.1.a System Description

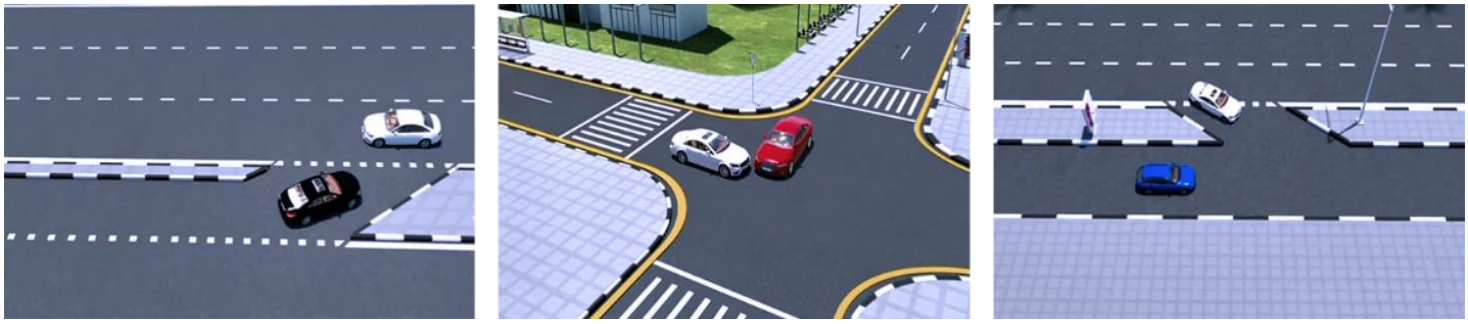


Figure 10.1 IMA Description

The problem of intersection traffic is very dangerous on man life. Every year, the Federal Highway Administration (FHA) reports approximately 2.5 million intersection accidents. Most of these crashes involve left turns [24].

• Nationally, 40 percent of all crashes involve intersections, the second largest category of accidents, led only by rear-end collisions [24].
Fifty percent of serious collisions happen in intersections and some 20 percent of fatal collisions occur there [24].
An estimated 165,000 accidents occur annually in intersections caused by red-light runners [24].
Fatalities caused by red-light runners run from 700-800 a year.
Making Intersections Safer

• Many strategies include geometric design and traffic control devices such as signals, signs, and markings.
A combination of these different strategies is required to solve the issue.
• Intersection safety is a top priority in the local, state, and national levels [24].

Many organizations such as the National Highway Transportation Safety Administration, Federal Highway Administration, Institute of Transportation Engineers and other public and private sectors continue to develop resources to make intersections safer [24].

• We can solve the problem using V2V (vehicle to vehicle) communication technology using some modules including sensors, MCUs and actuators[24].
We will use radar, GPS, ultrasonic, lcd, speaker, DSRC comm-protocol [24].

• Implementing some algorithms for different scenarios will solve the problem reducing or even preventing traffic intersection death cases keeping transportation better and safe for people precious life [24].
All this to determine the intersection points in the traffic lanes and alert the driver to take quick action before a disaster occurs [24].

10.1.b Main Algorithm

In the main algorithm we are focused on whether my car is going to cause any possible IMA Scenarios for any nearby car [24].
If yes, this algorithm will be ended by sending the possible scenario's ID to the other car [24].

- Flow Chart

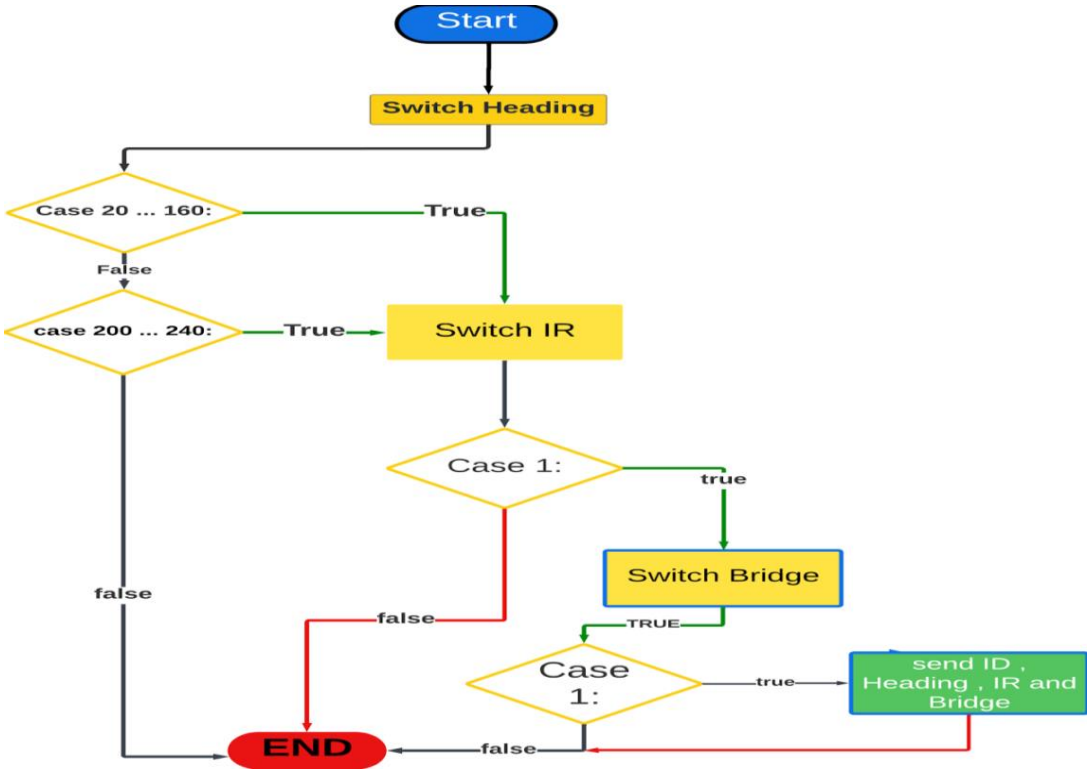


Figure 10.2 Flow Chart of Main Algorithm.

- Pseudo Code

```
Begin

  test heading
  if false end
  if intersection case true
  test IR case
  if false end
  if true
  test bridge case
  if false end
  if true
  send ID , IR ,HEADING and BRIDGE cases

END
```

Figure 10.3 Pseudo Code of Main Algorithm.

- Block Diagram

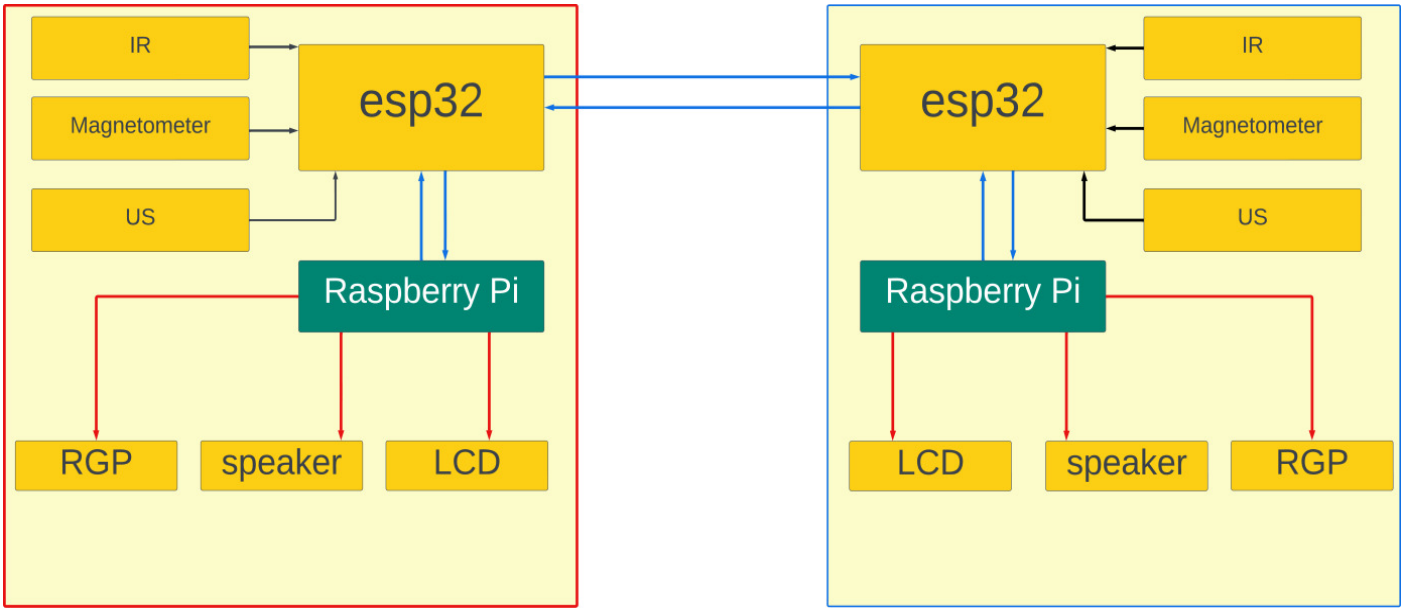


Figure 10.4 Block Diagram of Main Algorithm.

10.1.c ID Algorithm

In The ID Algorithm we focus on the system itself, we already know the ID of the 2 systems (EEBL &FCW) so from the direction we decide which system is happening now [24].

- **Flow Chart**

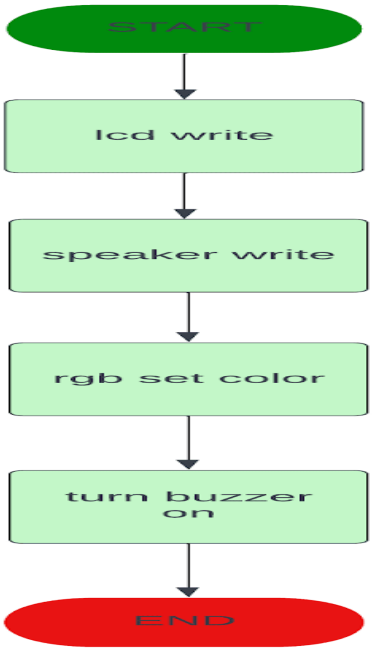


Figure 10.5 Flow Chart of ID Algorithm

- **Pseudo Code**

```
Begin

  write on LCD
  set rgb led color
  buzzer beeping once
  send speaker message

END
```

Figure 10.6 Pseudo Code of ID Algorithm

10.1.d IMA Layered Architecture

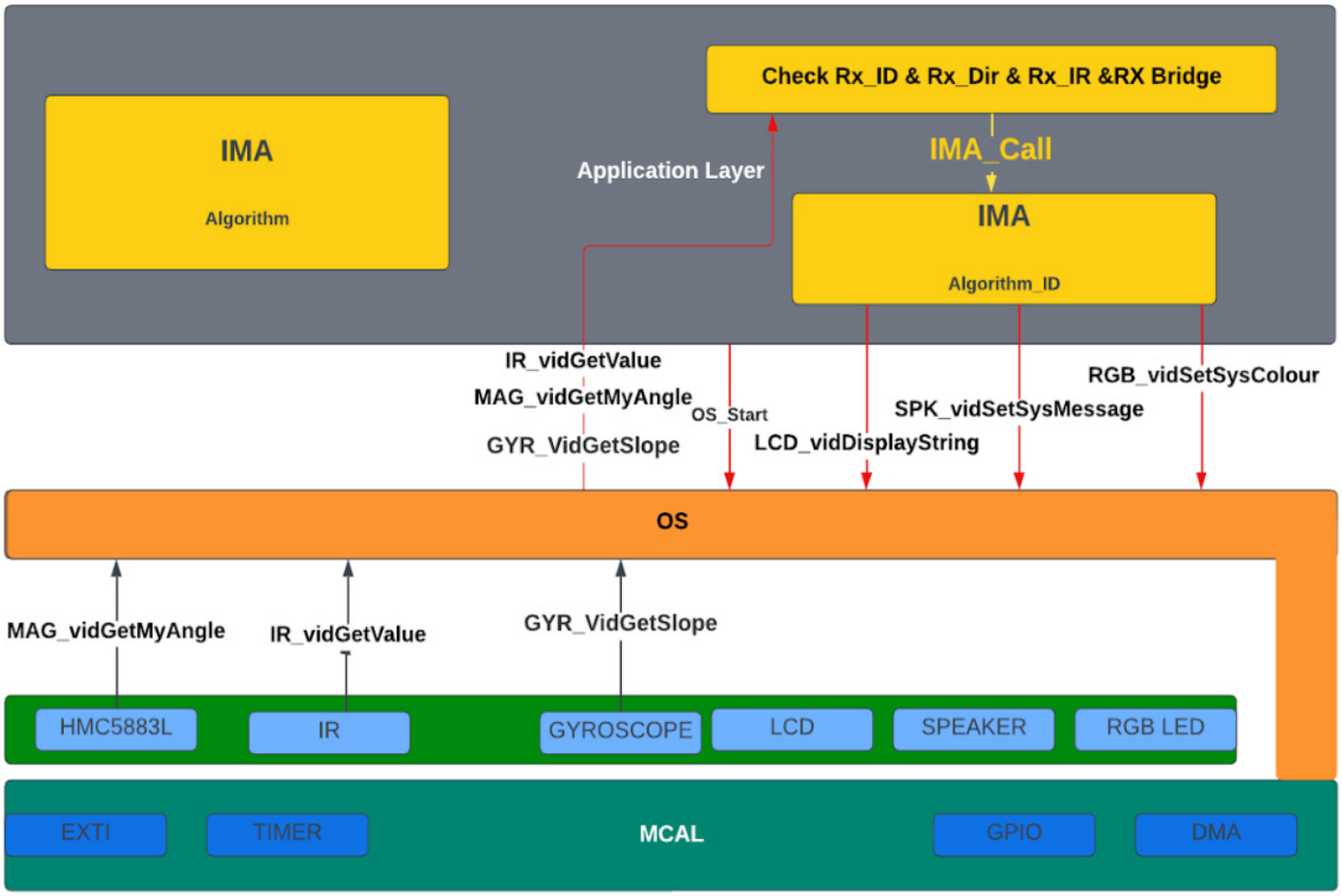


Figure 10.7 IMA Layered Architecture.

10.1.e IMA APIs Description

LCD_vidDisplayString

API Description	**Responsible for display string.**
API Input	Pointer to array of characters.
API return	Nothing.

RGB_vidSetSysColour

API Description	Blink it with the current subsystem predefined color.
API Input	Current subsystem ID.
API return	Nothing.

MAG_vidGetMyAngle

API Description	Get the direction angle of my car.
API Input	None.
API return	Nothing.

SPK_vidSetSysMessage

API Description	Send voice message to the speaker using DMA.
API Input	Current subsystem ID.
API return	Nothing.

IMA_vidAlg

API Description	Responsible for checking if there is a possible IMA Scenario caused by my car.
API Input	Nothing.
API return	Nothing.

IMA _vidAlgId

API Description	Responsible for taking action in case there is an intersection detected.
API Input	Nothing.
API return	Nothing.

EMBEDDED LINUX

CONTENT :

Chapter11: Embedded Linux
Chapter12: Embedded Linux Role in V2V Collision Avoidance

Embedded Linux

11

In this chapter, we will talk about Linux and its features. Also, will talk about Embedded Linux, its Architecture, and explain its components individually. Finally, we will explain what the Build systems is and explain Yocto in detail.

11.1 Introduction

Nowadays, Technology development increases rapidly, so we need applications morecomplicated with a huge number of features.

To deal with these expectations we needpowerful tools, like the Linux operating system to control these complex applicationsof embedded systems devices.

Ten years ago, embedded processors had limitations and didn't have the same capabilities as today's processors.

Now microcontrollers or systems on chip (SOC) generally provide interfaces with high application and suitable memory size and highspeed.

Undoubtedly, Embedded Linux will be applied to more and more devices.
Linux provides the flexibility and power of open-source development and offers greater security. Also, in a bare-metal environment, we are limited to a single application image. As we build out the application, we'll notice things get hard if oursystem must do a few totally different things simultaneously.

If we're developing for Linux, we can break this functionality into separate processes, where we can develop,debug, and deploy separately as separate binary images. So, let's know more detailsabout Linux [25].

11.2 Linux

11.2.1 Linux History

In the late 1960s, Bell Labs was involved in a project with MIT and General Electricto develop a time-sharing system, called Multiplexed Information and Computing Service (Multics). Multics wasn't a good success, so Bell labs decided to pullout [26].

In 1969, a team of programmers in Bell Labs who worked in Multics decided to makea better OS, they called it Uni-plexed Information and Computing Service (UNICS).
The team was led by great programmers:

i. **Ken Thompson.**
ii. **Dennis Ritchie.**

In 1971, the First Edition of Unics was released. In 1973, Ritchie rewrote B and called the new language the C language. In 1975, Unix V6 became very popular. Unix V6 was free and was distributed with its source code to many universities. Hence, manyorganizations had modified the Unix source code to build their own distributions. The most popular distributions is Berkeley Software Distribution (BSD).

In 1983, Bell Labs found that there are many distributions of Unix that are uncontrolled and untracked. They decided to make a closed commercial version of Unix and called it System V.

In 1983, Richard Stallman started the development of 100% free software. Not 90%and not 99.5%. It is totally free. Free does not mean free in cost, but free from freedom! It means that this system would give the user:

1. The freedom to run the program as you wish.
2. The freedom to copy the program and give it away to your friends and co-workers.
3. The freedom to change the program as you wish, the freedom to distribute an improved version and thus help build thecommunity.

GNU is a recursive acronym meaning GNU's Not Unix, A Unix-like operating system includes a kernel, compilers, editors, text formatters, mail software, graphical interfaces, libraries, games, and many other things. Thus, writing a whole operating system is a very large job. Stallman started in January 1984.

One of these most important components is the GNU C Compiler GCC. By 1990 wehad either found or written all the major components except one kernel.

In 1991, Linus Torvalds build a free, open-source kernel named Linux. Combining Linux with the almost-complete GNU system resulted in a completeoperating system: **GNU/Linux system** [26]. Estimates are that tens of millions of people now use GNU/Linux systems.

Nowadays, there are many distributions of GNU/Linux systems such as:

Ubuntu	Gentoo	CentOS	Debian	Linux mint	Fedora

11.2.2 Why Linux?

Linux is like other operating systems such as Windows, macOS, or IOS. Like them, Linux can have a graphical interface and the types of desktop software that you are used to, such as word processors, photo editors, video editors, etc. But Linux also differs in many important ways.

First, and perhaps its most important feature, it is open-source software.
The code used to create Linux is free and available for the public to view, edit, and for users to contribute to it.

Although the core pieces of theLinux operating system are generally widespread, many Linux distributions include different software options. That means that Linux is incredibly customizable.

Linuxis present in the software of many devices that we use daily. Linux is secure, flexible,and can receive excellent support from a large community of users. Linux is Multitasking and Multiuser.

In Linux, you get superuser access and privileges (for real), as Linux OS will not take any step without the consent of the superuser.

During Linux installation, you don’t need to install drivers for Wi-Fi, Bluetooth, mouse, touchpad, etc. explicitly as they can be installed during installation with littlepatience.

The Linux community is loyal to all the Linux users so it would give long Term Support.

Linux is ported on different platforms and is platform independent.
For other OSs, usually, a user would haveto go to the manufacturer's website to get driver support for different types of hardware.

The Linux kernel supports most of the hardware automatically via plug- and-play (largely in part because of the open-source community).
Some manufacturers also develop Linux versions of their proprietary drivers which could be easily installed via the software repository of distribution or by manually installing the provided binaries.

For these and other reasons, we are seeing an accelerated adoption rate of Linux in many Applications [26].

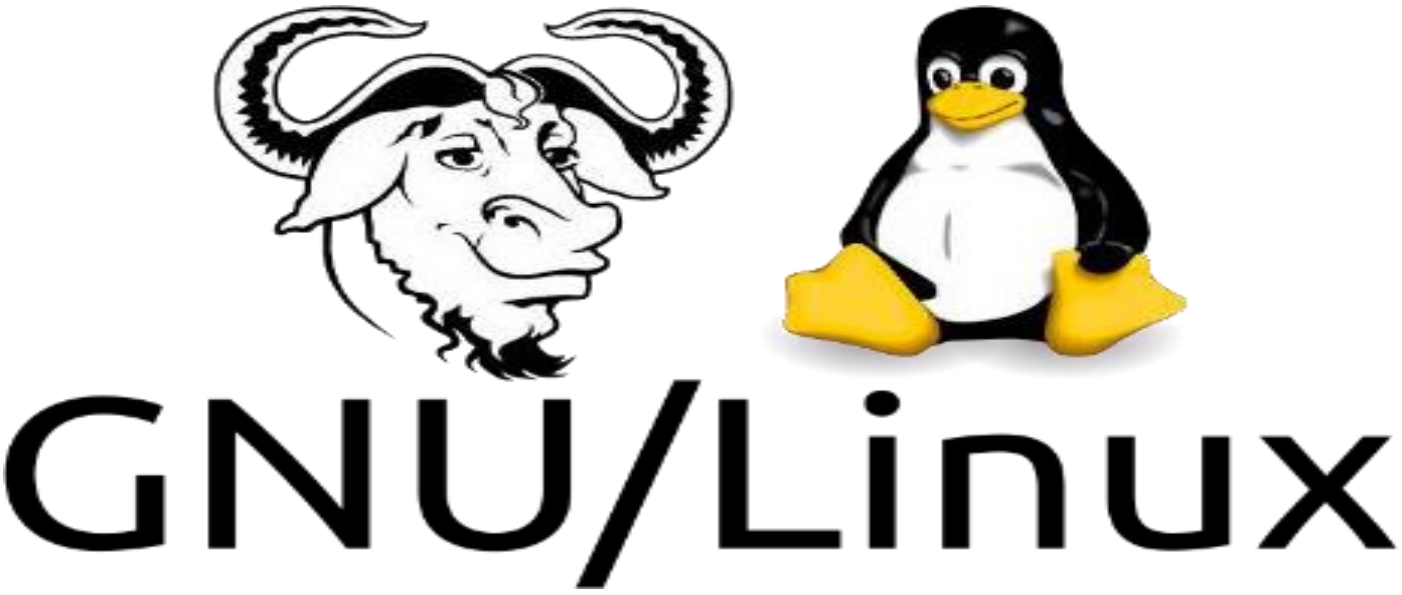


Figure 11.1 GNU/Linux OS

11.2.3 The difference between Linux and Windows

	Linux	Windows
Open Source	Linux is Open Source and is free to use.	Windows is not open source and is not free to use.
Case sensitivity	Linux file system is case-sensitive.	Windows file system is case insensitive.
kernel type	Linux uses monolithic kernel.	Windows uses a microkernel.
Efficiency	Linux is more efficient in operations as compared toWindows.	Windows is less efficient in operations.
Path Separator	Linux uses a forward slash as a path separatorbetween directories.	Windows uses backward slash as a path separator.
Security	Linux is highly secure ascompared to Windows.	Windows provides less security as compared to Linux.

Table 11.1 Difference between Linux and Windows.

11.3 Embedded Linux

Embedded Linux is a type of Linux operating system/kernel that was designed to beinstalled and used in embedded devices or systems. An embedded system is a set of computer hardware and software based on a microcontroller or microprocessor, controlled by a real-time operating system or RTOS, with limited memory, and that can vary both in size and complexity.

Although it uses the same kernel, embedded Linux is quite different from the standard operating system. First, it gets Customized for embedded systems and, therefore, is much smaller in size, requires less processing power, and has minimal features.

11.4 Embedded Linux Architecture

Every Embedded Linux project begins by obtaining, customizing, and deployingthese Elements (Components):

1. Toolchain
2. Bootloader
3. Kernel
4. Root filesystem
5. Specific Application

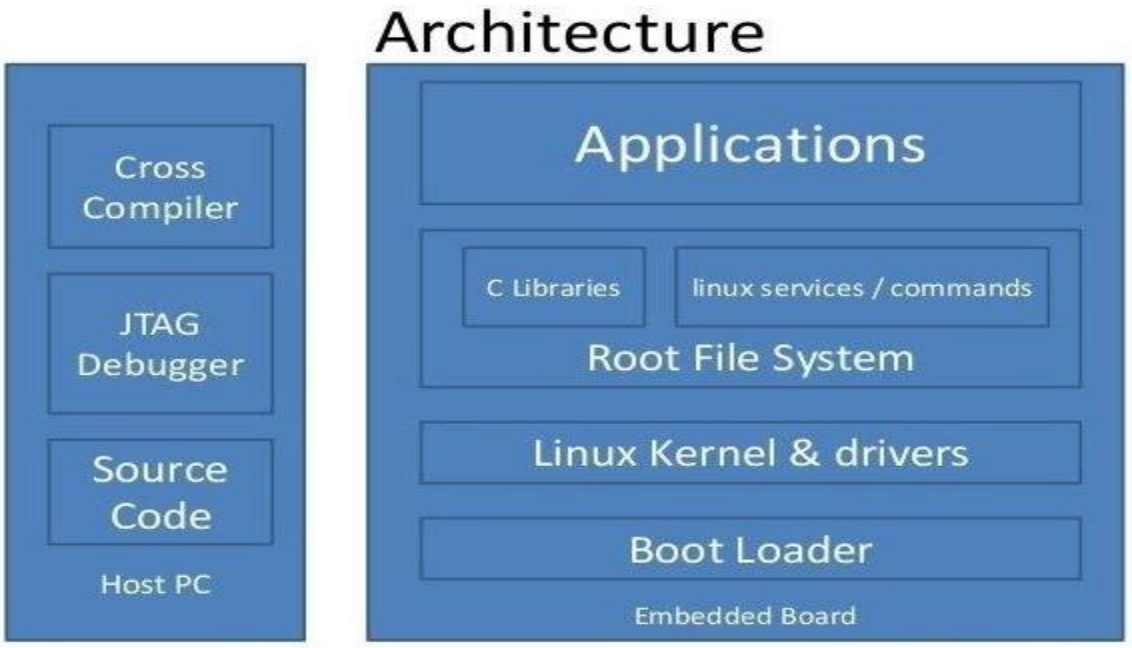


Figure 11.2 Embedded Linux Architecture.

The embedded system build process is usually done on the host PC using cross-compilation tools. Because target hardware does not have enough resources to run tools that are used to generate a binary image for target embedded hardware. The process of compiling code on one system (host system) and generating source code runs on the other system is known as cross-compilation [28].

11.4.1 Toolchain

A toolchain is a set of tools that compiles source code into executables that can run on the target device and includes a compiler, a linker, and run-time libraries. Initially, need one to build the other three elements of an embedded Linux system: the bootloader, the kernel, and the root filesystem. It has to be able to compile code written in assembly, C, and C ++ since these are the languages used in the base open-source packages [28].

There are two types of toolchain:

- Native toolchain: This toolchain runs on the same type of system, sometimes the same actual system, as the programs it generates. This is the usual case fordesktops and servers.
- Cross toolchain: This toolchain runs on a different type of system than the target, allowing the development to be done on a fast desktop PC and then loaded onto the embedded target for testing.

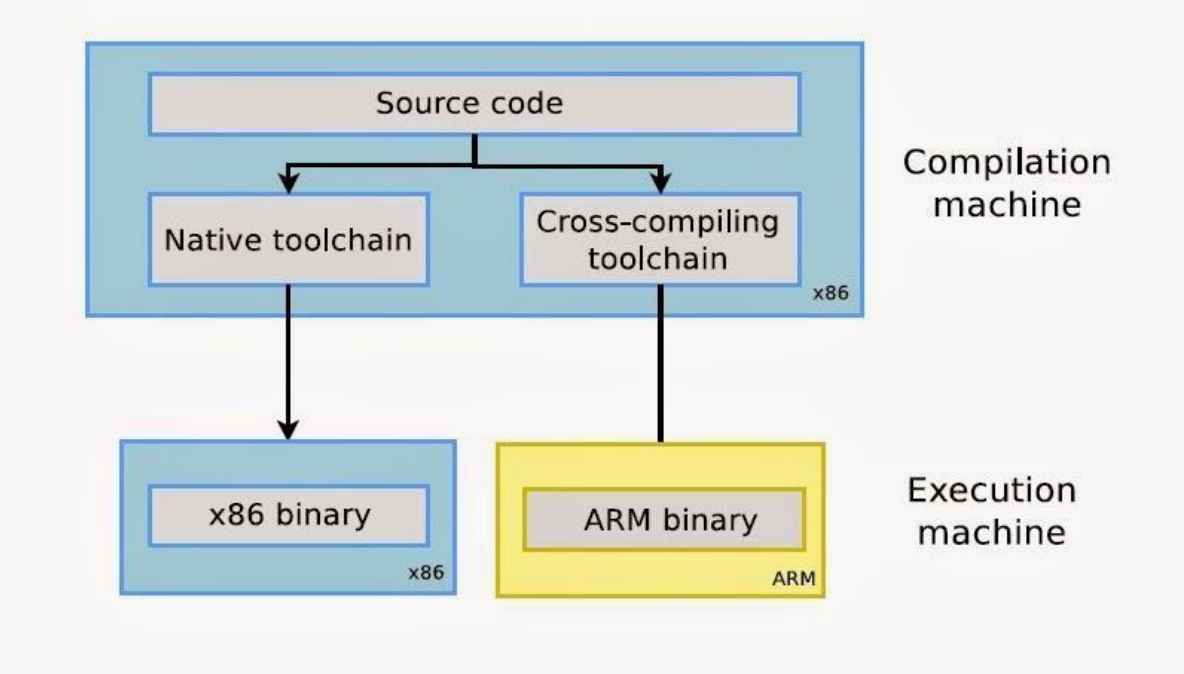


Figure 11.3 Types of Toolchains

Almost all embedded Linux development is done using a cross-development toolchain, partly because most embedded devices are not well suited to program development since they lack computing power, memory, and storage, but also because it keeps the host and target environments separate[29].

Toolchain consists of:

- Binutils: is a set of tools to generate and manipulate binaries for a given CPU architecture. ex: as, ld, ar, and ranlib.

- Kernel headers: The C library and compiled programs need to interact with thekernel. For example, to get available system calls and their numbers or for available data structures. That's why compiling the C library requires kernel headers, and many applications also require them.

- GCC: GNU Compiler Collection, the famous free software compiler. Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, and generate code for a large number of CPU architectures, including ARM, AVR, Black n, CRIS, FRV, M32, MIPS, MN10300, PowerPC, SH, v850, i386, x86 64, IA64,and Xtensa.

- C library: The C library is a very important component of a Linux system Interface between the applications and the kernel. It provides the well-known standard C API to ease application development. Several C libraries are available: glibc, uClibc, eglibc, dietlibc, newlib, etc. The choice of the C library for our toolchain must be made at the time of the cross-compiling toolchain generation.

There are three choices for our cross-development toolchain: a ready-built toolchain that matches our needs, use the one generated by an embedded build tool, selecting a Build System, or can create one using crosstool-NG [28].

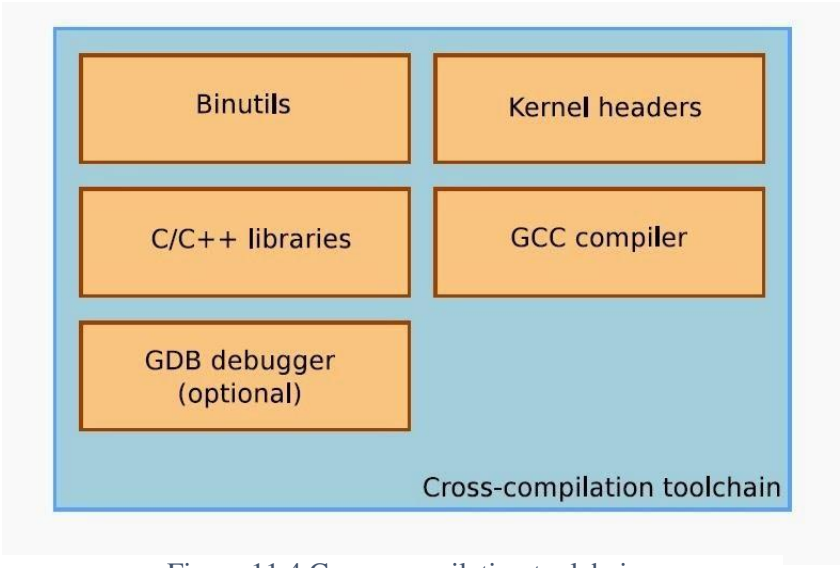


Figure 11.4 Cross-compilation toolchain

11.4.2 Bootloader

The bootloader has two main jobs: basic system initialization and the loading of the kernel. In fact, the first job is somewhat subsidiary to the second in that it is only necessary to get as much of the system working as is needed to load the kernel. Whenthe computer is powered on, after performing some initial setup, it will load a bootloader into memory and run that code. The bootloader's main job is to find the operating system's binary program, load that binary into memory, and run the operating system. In our case, this is the Linux kernel. The bootloader is done at thispoint, and all of its code and data in RAM are usually overwritten by the operating system. The bootloader won't run again until the computer is reset, or power cycled again [30].

The bootloader in embedded systems is different from a typical laptop, desktop, or server computer. A typical PC usually boots into what we call the BIOS first and thenruns Grub as the bootloader. Embedded Linux systems boot using Das-UBoot or U-Boot for short as the bootloader. So, Let's Talk about boot sequence firstly for Linuxmachines in general then for Raspberry pi.

Linux machine booting sequence: the time between system power on and login is called the booting time, there are 6 high-level stages of a typical Linux boot process.

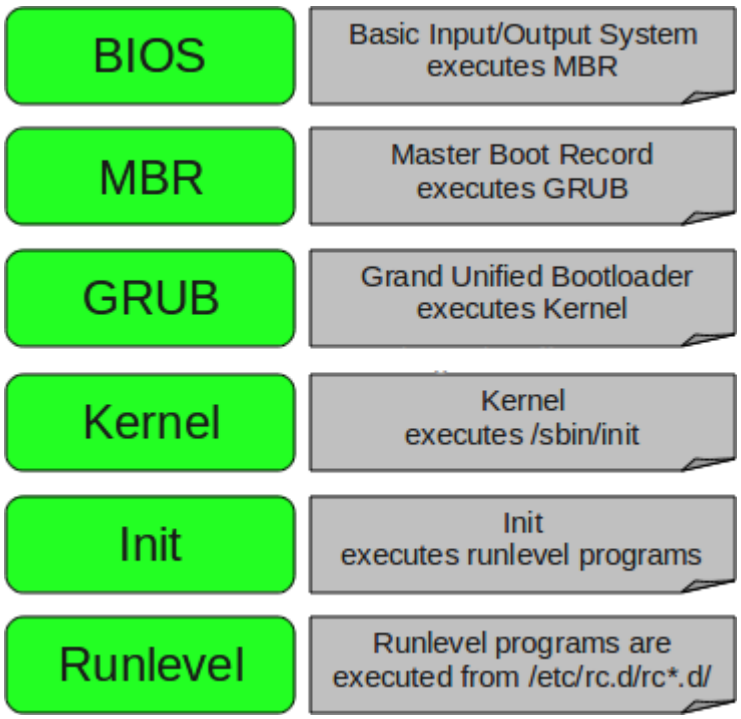


Figure 11.5 Booting Sequence

BIOS

- BIOS stands for Basic Input/Output System
- Performs some system integrity checks
- Searches, loads, and executes the boot loader program.
- It looks for boot loader in floppy, CD-ROM, or hard drive. You can press a keyduring the BIOS startup to change the boot sequence.
- Once the boot loader program is detected and loaded into the memory, BIOSgives the control to it.
- So, in simple terms, BIOS loads and executes the MBR boot loader.

MBR

- MBR stands for Master Boot Record.
- It is in the 1st sector of the bootable disk. Typically, /dev/had.
- MBR is less than 512 bytes in size. This has three components 1) primary bootloader info in 1st 446 bytes 2) partition table info in next 64 bytes 3) mbr validation check in last 2 bytes.
- It contains information about GRUB.
- So, in simple terms MBR loads and executes the GRUB boot loader.

GRUB

- GRUB stands for Grand Unified Bootloader.
- GRUB is the typical boot loader for most modern Linux systems.
- If there are multiple kernel images installed on the system, we can choose whichone to be executed.
- So, in simple terms GRUB just loads and executes Kernel [31].

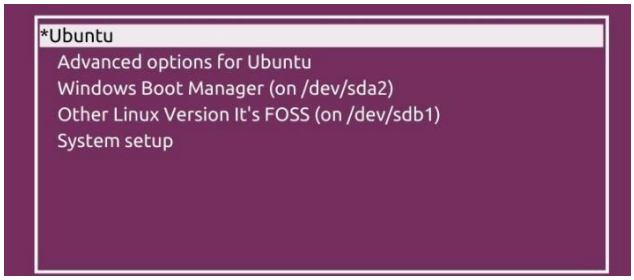


Figure 11.6 GNU GRUB

Kernel

- The kernel is often referred to as the core of any operating system, Linuxincluded.
- It has complete control over everything in your system.
- mounts the root file system that's specified in the grub.conf file.
- it executes the /sbin/init program, which is always the first program to be executed. You can confirm this with its process id (PID), which should always be 1.
- The kernel then establishes a temporary root file system using Initial RAM Disk (initrd) until the real file system is mounted.

Init

- system executes runlevel programs.
- Looks at the /etc/inittab file to decide the Linux run level.
- Following are the available run levels
  - 0 – halt
  - 1 – Single user mode
  - 2 – Multiuser, without NFS
  - 3 – Full multiuser mode
  - 4 – unused
  - 5 – X11
  - 6 – reboot
- Init identifies the default initlevel from /etc/inittab and uses that to load allappropriate programs.
- Execute ‘grep initdefault /etc/inittab’ on your system to identify the default runlevel.
- systemd will then begin executing runlevel programs [32].

Runlevel programs

- When the Linux system is booting up, you might see various services getting started. For example, it might say “starting sendmail …. OK”. Those are the runlevel programs, executed from the run level directory as defined by your runlevel.

- Depending on your default init level setting, the system will execute theprograms from one of the following directories.
    - Run level 0 – /etc/rc.d/rc0.d/
    - Run level 1 – /etc/rc.d/rc1.d/
    - Run level 2 – /etc/rc.d/rc2.d/
    - Run level 3 – /etc/rc.d/rc3.d/
    - Run level 4 – /etc/rc.d/rc4.d/
    - Run level 5 – /etc/rc.d/rc5.d/
    - Run level 6 – /etc/rc.d/rc6.d/
- Under the /etc/rc.d/rc*.d/ directories, you would see programs that start with Sand K.
- Programs that start with S are used during startup. S for startup.
- Programs start with K are used during shutdown. K for kill.
- There are numbers right next to S and K in the program names. Those are the sequence number in which the programs should be started or killed.
- For example, S12syslog is to start the Syslog daemon, which has the sequence number 12. S80sendmail is to start the Sendmail daemon, which has thesequence number 80. So, the Syslog program will be started before Sendmail [25].

As explained previously the bootloader in embedded systems is different from atypical laptop, Let's Know about the Raspberry pi boot sequence.

1. When the Raspberry Pi is first turned on, the ARM core is off, and the GPU core is on. At this point the SDRAM is disabled.
2. The GPU starts executing the first stage bootloader, which is stored in ROM onthe SoC. The first stage bootloader reads the SD card loads the second stage bootloader (bootcode.bin) into the L2 cache and runs it.
3. bootcode.bin enables SDRAM, reads the third stage bootloader (loader.bin) from the SD card into RAM, and runs it.
4. loader.bin reads the GPU firmware (start.elf).
5. start.elf reads config.txt, cmdline.txt, and kernel.img [32].

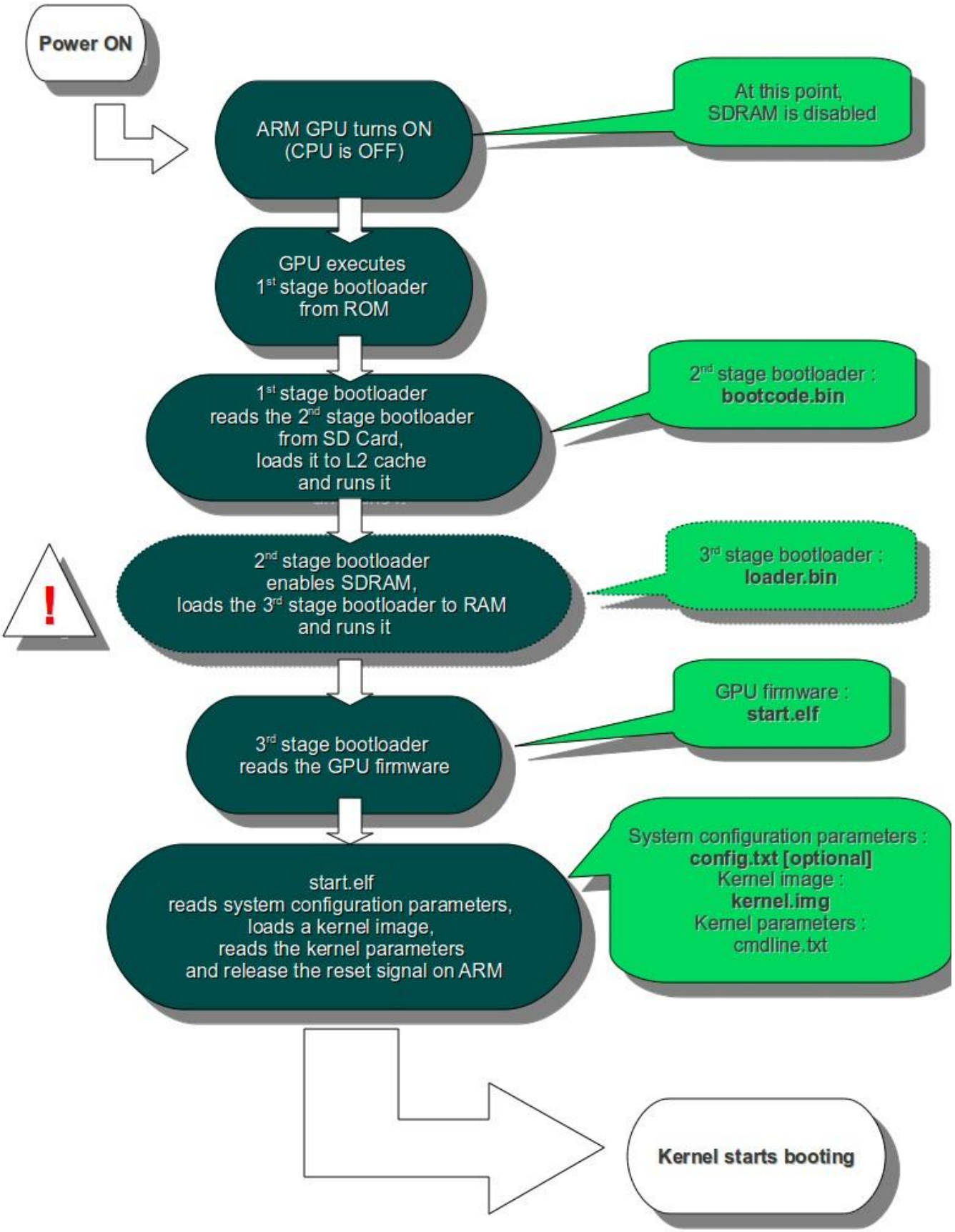


Figure 11.7 Raspberry pi boot sequence

After explaining the booting sequence, let's know how to choose a bootloader. Bootloaders come in all shapes and sizes. The kind of characteristics we want from a bootloader is that they are simple and customizable with lots of sample configurations for common development boards and devices.
The following table shows a number of them that are in general use:

Name	Architectures
Das U-Boot	ARM, Blackfin, MIPS, PowerPC, SH
Barebox	ARM, Blackfin, MIPS, PowerPC
GRUB 2	X86 , X86_64
RedBoot	ARM, MIPS, PowerPC, SH
CFE	Broadcom MIPS
YAMON	MIPS

Table 11.2 Architecture of a different bootloaders

11.4.3 Kernel

The kernel is the component that is responsible for managing resources and interfacing with hardware, and so affects almost every aspect of our final software build. It is usually tailored to our hardware configuration.

Once the bootloader loads the Linux kernel into memory and runs it, the Kernel will begin running its startup code. This startup code will initialize the hardware, initialize system critical data structures, initialize the scheduler, initialize all the hardware drivers, initialize filesystem drivers, mount the first filesystem, and launch the first program, among other things.

The Linux kernel's main job is to start applications and provide coordination among these applications (or programs, as they're usually called in Linux). The Linux kernel doesn't know about all programs that are supposed to run. So the Linux kernel starts only one program and lets that program launch all the other programs that are needed. This very first program is called the init program, or sometimes just "init" for short. If the kernel can't find the init program, the kernel's purpose is gone and the kernel crashes.

The main difference in the Linux kernel for embedded systems is that it is built to run on a different CPU architecture. Otherwise, the way the kernel operates is consistent with a typical PC, which is one of its strengths [28].

Applications running in User space run at a low CPU privilege level. They can do very little other than making library calls. The primary interface between the User space and the Kernel space is the C library, which translates user-level functions, such as those defined by POSIX, into kernel system calls.

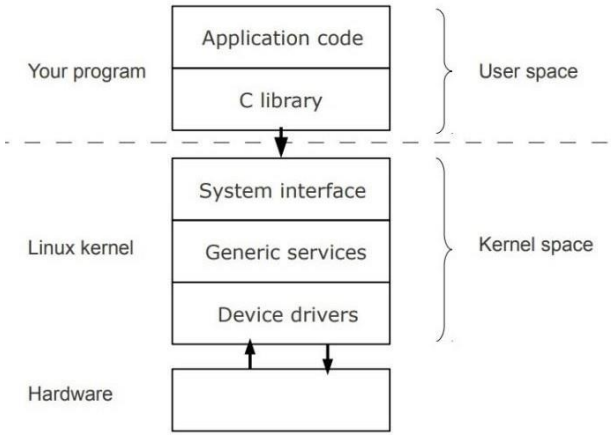


Figure 11.8 Kernel main jobs

The system call interface uses an architecture-specific method, such as a trap or a software interrupt, to switch the CPU from low privilege user mode to high privilege kernel mode, which allows access to all memory addresses and CPU registers. The System call handler dispatches the call to the appropriate kernel subsystem: memory allocation calls go to the memory manager, filesystem calls to the filesystem code, and so on. Some of those calls require input from the underlying hardware and will be passed down to a device driver. In some cases, the hardware itself invokes a kernel function by raising an interrupt.

To choose the kernel for the project, Balancing the desire to always use the latest version of software against the need for vendor-specific additions and an interest in the long-term support of the code base is required [28].

11.4.4 Root Filesystem

In Linux, the kernel loads programs into memory separately, and the kernel expects these programs to be stored on some medium organized into files and directories. This organization of files and directories is called a filesystem. As is true of many operating systems, Linux has filesystems on media, the data actually stored on a storage medium, and filesystem drivers, the code that knows how to interpret and update the filesystem data on the medium.

In Linux, this medium is often a hard disk. However, embedded systems often don’t have a hard drive, so the medium can be other hardware devices like SD cards, flash memory, or even RAM to name a few.

During Linux startup, the topmost filesystem gets associated with (or mounted to) this directory, and all the contents of that filesystem appear under /. This topmost filesystem is called the root filesystem.

Linux systems expect the root filesystem to be laid out a certain way. So this filesystem is special and can't just be some random set of directories and files. This is where directories like bin, sbin, etc, and more come from.

The main point here is that Linux looks for this first program, this init program, to reside in the filesystem. The root filesystem needs to be created in advance and be mounted to '/' before the kernel can launch the init program.
Because embedded systems have different hardware constraints, often Linux embedded systems use special filesystem formats rather than the typical EXT3, EXT4, btrfs, or xfs used on desktop or laptop computers[34].

The first objective is to create a minimal root filesystem that can give us a shell prompt. Then using that as a base we will add scripts to start other programs up, andto configure a network interface and user permissions. Knowing how to build the rootfilesystem from scratch is a useful skill.
To make a minimal root filesystem, we these components:

- init
- Shell
- Daemons
- Shared libraries
- Configuration files
- Device nodes
- Kernel modules

11.4.5 Specific Application

When the kernel finds, loads and runs the init program, that program then is responsible for bringing up the rest of the system.
The init program is also responsible for starting regular programs. These programs do have user interaction. Embedded systems often have just a few user programs, sometimes just one. In an embedded system, this set of programs makes the device do what it's supposed to do [33].

11.5 Build Systems

As explained previously, an embedded Linux system has several major components,all of which are derived from freely available source code and all of which may require varying levels of customization.

We can build individually a toolchain, a bootloader, a kernel, and a root filesystem, and then combine them into a basic embedded Linux system. It has the advantage that we are in complete control of the software, and we can tailor it to do anything we like. If we want it to do something truly odd but innovative, or if we want to reduce the memory footprint to the smallest size possible, building our own is the way to go.

But, in most situations, building manually is a waste of timeand produces inferior, unmaintainable systems. The idea of a build system is to automate all the steps of building the Embedded Linux major components. A capable embedded Linux build system helps us create an embedded Linux distributiontailored to our unique requirements.

This must include a cross-tool chain and all the packages required for our project. our build system should be able to generate root file systems in our choice of binary formats, our embedded Linux kernel image with our configuration, a bootloader image, and any other necessary files and utilities so that these can be properly deployed.

There are several build systems available, eachhas its own ideologies and focuses on building the Linux operating system for the target. Below given are the most popular among them:

11.5.1 Buildroot

This is a build system based on make. This is simple as the GNU build system. Lightweight build system, easy to edit and configure using menu configuration. It supports building several embedded applications with minimal footprint.

11.5.2 Yocto / OpenEmbedded

This build system is based on bitbake instead of make. Supports a huge list of packages. Supports several target boards and architectures. Supports a Layered approach to segregate and maintain the build scripts.Became a de-facto build system for embedded Linux.

11.5.3 OpenWRT / LEDE

This build system is a popular build system to build custom distributions for networking devices like routers and access points. It is a derivative of buildroot. Has its own lightweight software components as a replacement for the init system(procd), the network manager(netifd), interprocess communication(ubus), etc. it Has seven network-relatedated software components natively supported. It is also the base of a few other distributions like QSDK for Qualcomm network SoCs.

11.5.4 AOSP / Soong

The Android Open Source Project has built its own build system called Soong which is based on the blueprint (metabuild system) and Kati (Modified Make) and ninja (alternative for make). Soong builds an Android system with lots of Android-specific components like its own init system, HAL codes, system services, Java packages, and external opensource packages.

11.6 The Yocto Project

The Yocto Project is more general in the way it defines the target system, so it can build complex embedded devices. Every component is generated as a binary package, by default, using the RPM format, and then the packages are combined to make the filesystem image. Furthermore, you can install a package manager in the filesystem image, which allows you to update packages at runtime. In other words, when you build with the Yocto Project, you are, in effect, creating your own customLinux distribution.

The Yocto Project is primarily a group of recipes, similar to Buildroot packages but written using a combination of Python and shell script, together with a task scheduler called BitBake that produces whatever you have configured, from the recipes.

The Yocto Project is structured with cross-platform tools, and metadata, to enable developers rapidly to create customized Linux distributions from source code, whichsimplifies the development process.

Yocto Project has advantages in system and application development, archiving, and management. Developers can customize their systems in terms of speed, memory footprint, and even memory utilization. Yocto Project allows software customization and construction exchange for multiple hardware platforms and maintains a software stack in scale.

The Yocto Project collects several components, the most important of which are the following:

1. **OE-Core** is metadata composed of basic recipes, classes, and related files. These metadata are designed to be common in many different OpenEmbeddedderived systems (including Yocto projects).

2. **BitBake:** The core tool of the OpenEmbedded build system. BitBake plays the role of build a system engine and is responsible for parsing metadata, generating task lists from it, and then performing these tasks.

3. **Poky:** a reference distribution. Poky is the name of the reference distribution or reference OS of the Yocto Project. Poky includes the OpenEmbedded Build System (Bitbake and OpenEmbedded-Core) and a set of metadata to help us start building our own distribution. Poky uses OpenEmbedded Build System to build a small, embedded operating system. Poky is an integration layer on top of OE-Core. Poky provides the following:
    - A basic level of distro infrastructure to illustrate how to customize the distro.
    - A means to verify the Yocto Project components.

4. **Documentation:** This is the user's manuals and developer's guides for eachcomponent.

5. **Toaster:** This is a web-based interface to Bitbake and its metadata.

6. **ADT Eclipse:** This is a plugin for Eclipse.

The Yocto Project provides a stable base, which can be used as it is or can be extended using meta layers. When downloading the build system, the Poky build ‘file’ is calleda recipe and a layer. we can modify, assign, or anyway we need to create our own customized embedded Linux [28].

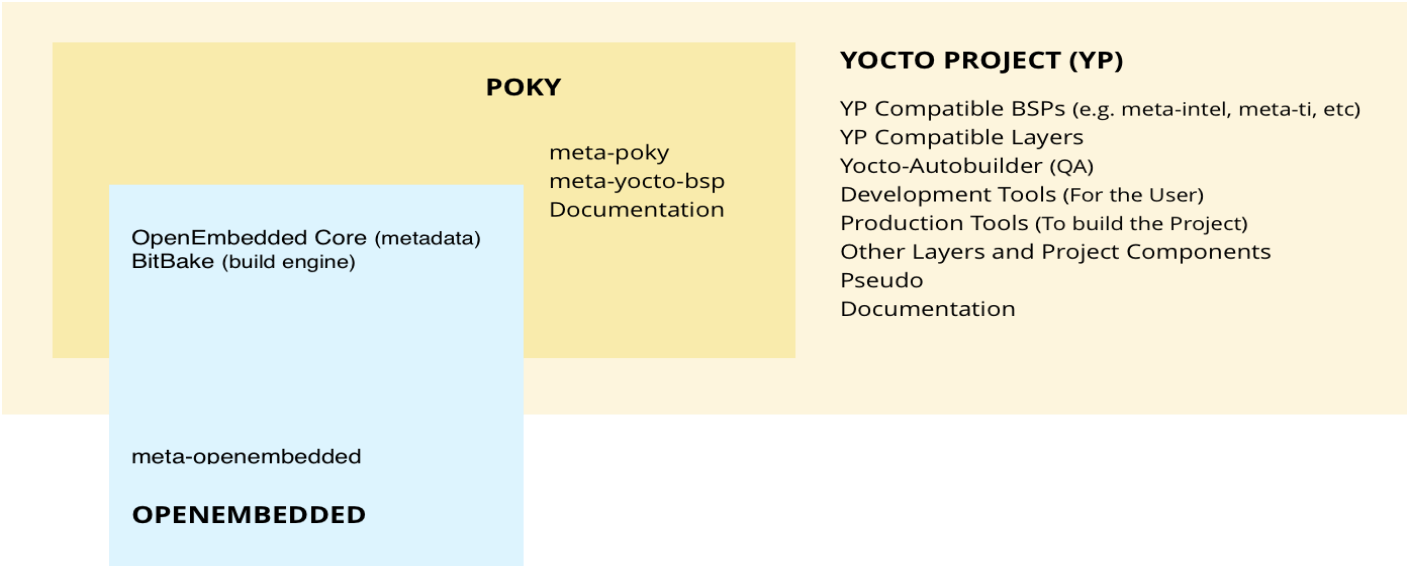


Figure 11.9 Yocto project components

Some Yocto Project Terminology:

1. **Recipe**: The most common form of metadata. Recipe contains a list of settings and tasks used to build a binary image file. Recipe describes where you get thecode and which (code) patch you need to apply. At the same time, Recipe also describes the dependencies on other recipes or libraries, as well as configuration and compilation options. Recipe is stored in Layer.

2. **Layer:** A collection of related recipes. A Layer is a repository containing relevant metadata that tells the OpenEmbedded build system how to build the target. Yocto Project's layer model promotes collaboration, sharing, customization, and reuse in the Yocto Project development environment. Layers logically separate the information of your project.

3. **Metadata:** Layer contains the recipe files, patches, and additional files providedby the user, other information referring to the build instructions, and data that controls what and how to build. A good example of the software layer might be the meta-Qt5 Layer from the OpenEmbedded Layer Index.

4. **A Board Support Package (BSP):** is a collection of information that defines how to support a particular hardware device, set of devices, or hardware platform. The BSP includes information about the hardware features present on the device and kernel configuration information along with any additional hardware drivers required.

5. **Configuration Files:** Files that hold global definitions of user-defined variables and hardware configuration information. They tell the build system what to build and put it into the image to support a particular platform.

6. **Packages:** The output of the build system used to create your final image [35].

11.7 Summary

Embedded hardware will continue to get more complex, following the trajectory set by Moore's Law. Linux has the power and the flexibility to make use of hardware in an efficient way. Linux is just one component of open-source software out of the many that you need to create a working product.

Linux is open-source software. The code used to create Linux is free and available for the public to view, edit, and for users to contribute to it. Linux is present in the software of a large number of devices that we use on a daily basis. Linux is secure, flexible, and can receive excellent support from a large community of users. Linux isMultitasking and Multiuser.
Embedded Linux is a type of Linux operating system/kernel that was designed to beinstalled and used in embedded devices or systems.

When an embedded computer starts, the Linux system will perform these steps:

- jump into the bootloader.
- jump into the kernel.
- mount the root filesystem.
- load and run init.
- load and run background services (or daemons)
- load and run applications.

Each of these steps invokes a component that is needed in the system.

Building embedded systems, which was a complex process earlier, has been simplified a lot by open-source build frameworks, and there exists a variety of choices and flexibility for embedded Linux developers.

Embedded Linux Role in V2V Collision Avoidance

12

Welcome to this chapter, where we embark on a crash course in Yocto and explore the fascinating realm of creating custom images. In the world of embedded systems and Linux-based development, Yocto Project has emerged as a powerful and flexible tool for building custom Linux distributions tailored to specific hardware platforms.

In this chapter, we will delve into the core concepts of Yocto Project, demystifying its components and workflow. We will explore the layers, recipes, and configurations that make up the backbone of Yocto, helping you understand the building blocks of your custom Linux image.

But our journey does not stop there. We will go beyond the basics and showcase the process of crafting our very own custom image using Yocto. This hands-on experience will give you a practical understanding of how to tailor a Linux distribution to meet your specific requirements, whether you're developing for an IoT device, an embedded system, or any other Linux-powered platform.
Throughout this chapter, we will guide you step by step, providing clear explanations, examples, and tips to ensure your success in mastering Yocto and custom image creation.

By the end, you will have the knowledge and confidence to tackle your own projects and create customized Linux images that perfectly align with your vision.
So, whether you're a seasoned developer looking to enhance your skills or a newcomer eager to dive into the world of embedded Linux, this chapter is your gateway to unlocking the full potential of Yocto Project and unleashing your creativity in building custom images. Let's embark on this exciting journey together!

12.1 Introduction to Yocto Project

Linux-powered systems have become an integral part of our lives, with many of us relying on them daily and some using them extensively. The Linux kernel serves as the backbone for various devices, including smartphones, tablet computers, personal computers, medical instruments, car infotainment systems, GPS navigation systems (now mostly replaced by smartphones), set-top boxes, Wi-Fi routers, and more. Additionally, numerous systems indirectly integrated into our daily lives, such as industrial gateway systems, telecommunication equipment, and global data servers, also utilize Linux kernels. It's worth noting that these systems employ different processors, ranging from powerful multi-core processors to simpler ones at their coreht [36].

The widespread adoption of Linux-powered systems suggests that there must be something that makes it convenient for manufacturers and designers to utilize Linux. Otherwise, an alternative system would have emerged. This is where the Yocto Project comes into play [36].

Before delving into the specifics of the Yocto Project, it's important to clarify what it is not:

- The Yocto Project is not an SDK (Software Development Kit) for your hardware machine; rather, it provides tools to build one.
- The Yocto Project is not a pre-built system binary image to deploy on your hardware machine; instead, it assists in building such an image.
- The Yocto Project is not a ready-made Linux distribution for your hardware machine; it facilitates the creation of a highly customized distribution tailored to resource-constrained hardware.

So, what exactly is the Yocto Project?

The Yocto Project is an open-source collaboration initiative that empowers developers to create custom Linux-based systems for embedded products, regardless of the hardware architecture. It offers a flexible set of tools and a collaborative platform for embedded developers worldwide to share technologies, software stacks, configurations, and best practices. These resources enable the creation of tailored Linux images for embedded devices [36].

The Yocto Project encompasses and maintains three essential development elements:

1). Integrated tools: These tools facilitate working with embedded Linux and include automated building and testing utilities, board support and license compliance processes, and component information for custom Linux-based embedded operating systems.

2). Poky: This is a reference embedded distribution that serves as a base-level functional distro, demonstrating how to customize a distribution. Poky also validates the Yocto Project and provides a starting point for customization. It is an integration layer built on top of oe-core.

3). OpenEmbedded build system: Co-maintained with the OpenEmbedded Project, this build system consists of foundation recipes, classes, and associated files that are widely applicable across various OpenEmbedded-derived systems, including the Yocto Project. It is a curated and quality-assured set of continuously validated recipes.

In simpler terms, oe-core comprises quality-assured foundational recipes used by Poky to generate a reliable and functional base Linux image.

Additionally, it's important to understand the concept of recipes within the Yocto project:

A recipe is a form of metadata that specifies instructions and settings for building packages. It defines source code locations, patches to apply, dependencies on libraries or other recipes, and configuration and compilation options.

The Yocto Project utilizes a layered approach, where recipes and configuration files are organized into layers. Each layer focuses on specific aspects, such as networking, applications, graphics subsystems, etc. This layered structure enhances scalability, versatility, and adaptability to different systems [36].

The BitBake tool plays a crucial role in the Yocto Project:

BitBake acts as a scheduler and execution engine that parses instructions (recipes) and configuration data. It creates a dependency tree to determine the order of compilation, schedules the compilation of code, and ultimately builds the specified custom Linux image (distribution) [36].

12.2 Setting up an Ubuntu Host

In this Section, we will discuss the process of configuring an Ubuntu host to perform Yocto builds. According to the Yocto Project documentation, Ubuntu is a supported Linux distribution on the host side, although certain versions have better support than others. It is recommended to use Long-Term Support (LTS) versions of Ubuntu as your host build PC. You can find the list of supported Linux distributions in the following section. Here, we will focus on setting up your PC running Ubuntu 20.04 LTS as the operating system.

Minimum Requirements

To ensure smooth Yocto builds, it is advisable to have at least 50 GB of free memory on your hard disk. The more free memory you have, the more future-proof your builds will be!

Software Tool Requirements [37].

There are several essential software tools required for performing Yocto builds on your Ubuntu 20.04 PC. These tools are common across different hosts, including Ubuntu 20.04. They are as follows:

1. Git: Git is used to fetch source code from various software repositories, as discussed in the previous chapter.

2. Tar: Most source code is downloaded as a torball to minimize data usage and speed up the fetch process.

3. Python3: Although Python3 is usually pre-installed on Ubuntu, it is recommended to verify that the installation is complete, and the version is equal to or higher than Python 3.6.0.

4. GCC: The GCC toolchain is utilized to handle the substantial number of builds involved in package generation and image generation processes.

To install these software tools, execute the following commands. For the second command, you may need to press the Enter/Return key when prompted by the package manager to fetch and install the packages [37].

```
$ sudo apt update
$ sudo apt install git git-lfs tar python3 python3-pip gcc
```

Ubuntu-Specific Package Requirements

In addition to the minimum requirements mentioned earlier, you need specific host packages to ensure successful Yocto builds. These packages are necessary for the steps (1 to 8) outlined in the previous chapter [37].

To install these packages, execute the command below. Press the Enter/Return key when prompted by the package manager to fetch and install the packages [37].

```
$ sudo apt install gawk wget git diffstat unzip texinfo gcc build-essential chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev pylint3 xterm python3-subunit mesa-common-dev zstd liblz4-tool
```

You may have noticed that the command above includes python3 and python3-pip as well. This is not an issue as the package manager will skip them if they are already installed on your system [37].

Optional Documentation Packages [37].

Yocto releases come with source documentation, and you have the option to generate your own documentation and include it using your meta layer. To build the Yocto documentation, you will need some additional packages [37].

To install these packages, execute the following command. Press the Enter/Return key when prompted by the package manager to fetch and install the packages [37].

```
$ sudo apt install make python3-pip
$ sudo pip3 install sphinx sphinx_rtd_theme pyyaml
```

Depending on your internet speed, the above steps may take between 5 to 15 minutes to complete. Once finished, you have successfully set up your Ubuntu 20.04 PC for your first Yocto build!

12.3 Build & Run your first Image

We dive right into the action and construct our inaugural Yocto image. We highly recommend undertaking this build regardless of the target machine you have in mind, as the resulting image is incredibly useful for quickly verifying the desired features [38].

- **Establishing a Working Directory**

Let's assume you're in your home directory.

1. The initial step involves creating a dedicated working directory for Yocto. While not mandatory, it is considered good practice to have a designated workspace rather than performing tasks randomly on your computer.

Execute the following commands to

```
$ mkdir yocto
$ cd yocto
```

2. Downloading Poky and Selecting the Desired Release

```
$ git clone git://git.yoctoproject.org/poky
$ cd poky
$ git checkout -b dunfell origin/dunfell
```

3. Creating Your Active Build Directory

To build directory serves as a container for all your Yocto builds. This allows you to neatly compartmentalize builds while reusing resources as much as possible. The build directory contains machine configurations, specific variables for your image or machine, additional recipes, and more [38].

```
$ source oe-init-build-env <name of your build directory>
```

If you don't specify a name, a build directory named "build" will be created automatically. When you want to use the same directory again, simply execute the same command. Additionally, once this command is executed, you will already be inside the newly created build directory.

- **Initiating Your First Build**

The initial build usually takes the longest as it involves downloading numerous tarballs from various upstream sources referenced in the recipes. Therefore, it's a good idea to break it down into two parts. The first step is to fetch all the source code, tarballs, etc. The second step involves performing the actual build using the downloaded resources, along with the configuration metadata and recipes [38].

Speaking of recipes, several core-image recipes are readily available for use. These core recipes enable you to create a functional Linux image without requiring any customizations for your platform. Successfully creating such an image validates that your build setup is operational, allowing you to proceed to more advanced builds for your specific machines [38].

A subset of these core images should already be visible if your last command is executed correctly, meaning you're already inside your build directory. Here are a few popular core images:

- core-image-base: A console-only image that fully supports the target hardware.
- core-image-minimal: A minimal image that allows the device to boot.
- core-image-sato: An image with Sato support, providing a mobile environment and visual style suitable for mobile devices.
- core-image-clutter: An image with support for the OpenGL-based Clutter toolkit.
- core-image-full-cmdline: A console-only image with a more comprehensive set of Linux system functionalities installed.

- ...

The default machine selected for Yocto build is named "qemux86-64." You can view this selection in the file "conf/local.conf." Qemu is a useful tool for testing Yocto images, as it allows you to verify the presence of all the desired features without downloading the image to your target machine. Additionally, it serves as a valuable Linux learning tool since Qemu runs the image using virtualization and has access to the same resources as your host machine (e.g., Ubuntu).

For this guide, we'll use the "core-image-sato" image and retain "qemux86-64" as the machine.

To fetch the necessary resources for this build, execute the following command:

```
$ bitbake core-image-sato --runall=fetch
```

Depending on your internet connection speed and CPU capabilities, this process may take anywhere from a few tens of minutes to an hour or more.

Once the resources have been fetched, it's time to initiate the build. Execute the following command:

```
$ bitbake core-image-sato
```

Depending on your CPU capabilities, the build process may take anywhere from a few tens of minutes to possibly a few hours (it took us 2.5 hours!).

- **Running the Image**

  Once the build is complete, you can find the output image in the "tmp/deploy/images/qemux86-64" folder. Poky provides a handy tool called "runqemu," which simplifies the steps required to run the image. It all comes down to executing a simple command to boot your very first Yocto image! To do so, run the following command:

  ```
  $ runqemu qemux86-64 nographic
  ```

  The "nographic" argument informs runqemu that we're not interested in launching a graphical user interface (GUI). This is useful as graphics card access can sometimes be finicky and lead to crashes.

12.4 Building basic image for RPI

We will explore how to advance our work from Part 3 by creating a test image for the Raspberry Pi. If you are unfamiliar with the Raspberry Pi, you can refer to this YouTube playlist for a crash course on the Raspberry Pi ecosystem and basic usage. [38].

Assuming that you have already created a build directory, this section will guide you through the process. If you haven't, please refer to this section and follow the steps provided, stopping after creating your build directory[38].

Are you ready? Let's begin building an image for the Raspberry Pi 4.

Setting up the build environment
Let's assume that your build directory is named 'build-rpi.' Execute the following command in your poky directory to set it up:

```
$ source oe-init-build-env build-rpi
```

NOTE: Replace 'build-rpi' with the name of your build directory if different.

Bitbake layer configuration for Raspberry Pi

When we built the image for QEMU, we didn't need to configure any layers as the poky distribution already includes the necessary recipes for the specific QEMU machine. For example, our machine in the previous build was 'qemux86-64.'

According to the readme.md document of the meta-raspberrypi layer, we need at least the meta-openembedded layer, and potentially more depending on the required features for the image. The recipes within meta-openembedded enable builds for embedded architectures such as ARM and MIPS. If you need a quick refresher on Open-Embedded, recipes, etc., you can check out this concise introduction to Yocto!

For our build, we will also add other layers essential for networking, Python, multimedia, and more. Let's download these layers first and then add them to our build configuration [38].

1. Initialize Directory
To maintain good practice, it's recommended to have the meta-layers in the same directory as 'poky' so that multiple build directories can utilize them.

In our system, the folder structure looks like this:
yocto
poky
...
<< newly downloaded layers will be placed here >>

Let's navigate to the yocto directory first.

```
$ cd ../..
$ pwd <your yocto directory>/yocto
```

2. Downloading Necessary files

```
$ git clone git://git.openembedded.org/meta-openembedded -b dunfell
```

3. Next, we'll download the meta-raspberrypi layer source code. This layer contains all the necessary recipes to build basic images for the Raspberry Pi boards. Here we can see the layering concept in action. The meta-raspberrypi layer includes recipes for the kernel, drivers, configuration, etc., which override those in the corresponding poky and meta-openembedded recipes. For instance, the Linux kernel configuration.

```
$ gitclone git://git.yoctoproject.org/meta-raspberrypi -b dunfell
```

4. Now we have downloaded the required layers, and our folder structure will look like this:

```
yocto
poky
...
meta-raspberrypi
...
meta-openembedded
meta-oe
meta-networking
meta-python
meta-multimedia
```

5. Adding layers to the build using bitbake commands
   While you can manually modify the bitbake layer configuration by editing the 'conf/bblayers.conf' file in your active build directory, this approach is prone to syntax errors that can cause failed builds due to parser failure.

   A better approach is to use the 'bitbake-layers' command. To add the necessary layers to the build, execute the following commands from

   your yocto directory:

   ```
   $ bitbake-layers add-layer ./meta-openembedded
   $ bitbake-layers add-layer ./meta-raspberrypi
   $ bitbake-layers add-layer ./meta-openembedded/meta-oe
   $ bitbake-layers add-layer ./meta-openembedded/meta-python
   $ bitbake-layers add-layer ./meta-openembedded/meta-networking
   $ bitbake-layers add-layer ./meta-openembedded/meta-multimedia
   ```
6. You can verify that your layer configuration is updated by executing the following command:

   ```
   $ bitbake-layers show-layers
   ```

- **Choosing the machine and the image**

Image recipes are .bb files that contain all the details needed to build an image for a specific machine. Machine names can be found in the respective meta- layers.

1. Selecting the machine
   You can find the machine names supported by the meta-raspberrypi layer in the 'meta-raspberrypi/conf/machine' directory. Each .conf file represents a machine configuration, and the machine's name matches the file's name (without the .conf extension). You can list the supported machines in meta-raspberrypi by executing the following command from the yocto directory:

   ```
   $ ls meta-raspberrypi/conf/machine
   ```
   In our case, we want to build an image for the Raspberry Pi 4, so we'll use the machine named 'raspberrypi4.'

2. Selecting the image
   To see the available images you can build, check the contents of the 'meta-raspberrypi/recipes-core/images' folder. Execute the following command from the yocto directory:

   $ ls meta-raspberrypi/recipes-core/images

   You should see 'rpi-basic-image.bb,' 'rpi-hwup-image.bb,' and 'rpi-test-image.bb.' The names of these files without the .bb extension represent the available images.

• Setting the machine variable

By default, the build target machine is set to 'qemux86-64.' We need to change this to 'raspberrypi4.' Open the 'conf/local.conf' file located in your build directory and navigate to the uncommented line that sets the machine variable. Comment it out by adding a # at the beginning, and then set 'raspberrypi4' as the machine. Here's how the final result should look:

MACHINE ??= "qemux86-64"
MACHINE ??= "raspberrypi4"
PRO-TIP: To test this Raspberry Pi 4 image, it's useful to enable the serial console. Add the following line towards the end of the 'conf/local.conf' file:

ENABLE_UART = "1"
The serial console on the Raspberry Pi 4 boards is available on pin numbers 8 (GPIO14 – TXD) and 10 (GPIO15 – RXD) of the 40-pin IO connector.

Now we're ready to start the build process. We'll follow the same approach as before, fetching the necessary resources first and then performing the build itself.

Fetching and Building!
To fetch the necessary resources without initiating the build, execute the following command:

$ bitbake rpi-test-image --runall=fetch

Depending on your internet connection and CPU capability, this process can take anywhere from a few minutes to tens of minutes.

Once the fetching is complete, you can start the build by executing the following command:

```
$ bitbake rpi-test-image
```

Depending on your internet connection and CPU capability, the build process can take anywhere from a few tens of minutes to a few hours. Patience is key!

Congratulations! If all goes well, you will have successfully built a test image for the Raspberry Pi 4 using Yocto!"

12.5 Creating and adding a new layer to your image

Covering a fundamental yet profoundly significant subject, we embark on a journey that often serves as the initial step towards personalizing your image. Our focus centers on the creation and integration of a new layer into your image [38].

Layers - A Recap

In the introductory section, we explored the layered nature of the Yocto build system and its ability to facilitate extensive software reuse, as well as seamless migration between different hardware platforms. Layers empower you to encapsulate your unique elements (referred to as "secret sauce" or "source code") within your own layer. Moreover, they allow you to override configurations for recipes present in lower-level layers. For instance, you can enhance the packages installed in an existing base image from meta-oe by incorporating your software or customize the Linux kernel configuration for a specific image, among other possibilities.

Now, let's delve into the process of creating our own layer.

Creating a New Layer

The recommended and straightforward approach for creating a new layer involves using the bitbake-layers script. Once you have sourced the oe-build-init-

env script within the poky/ directory, the bitbake-layers script becomes readily accessible[38].

The bitbake-layers script offers more than just layer creation. To explore its full capabilities, we first initialize our build environment and then consult the available help documentation. Execute the following commands from within the poky/ directory:

```
$ source oe-init-build-env
$ bitbake-layers --help
```

The output should resemble the following:

```
NOTE: Starting bitbake server...
usage: bitbake-layers [-d] [-q] [-F] [--color COLOR] [-h] <subcommand> ...

BitBake layers utility

optional arguments:
  -d, --debug           Enable debug output
  -q, --quiet           Print only errors
  -F, --force           Force add without recipe parse verification
  --color COLOR         Colorize output (where COLOR is auto, always, never)
  -h, --help            show this help message and exit

subcommands:
  <subcommand>
    add-layer           Add one or more layers to bblayers.conf.
    remove-layer        Remove one or more layers from bblayers.conf.
    flatten             Flatten layer configuration into a separate output directory.
```

show-layers	Show currently configured layers.
show-overlayed	List overlayed recipes (where the same recipe exists in another layer).
show-recipes	List available recipes, showing the layer they are provided by.
show-appends	List bbappend files and recipe files they apply to.
show-cross-depends	Show dependencies between recipes that cross layer boundaries.
layerindex-fetch	Fetch a layer from a layer index along with its dependent layers and add them to conf/bblayers.conf.
layerindex-show-depends	Find layer dependencies from the layer index.
create-layer	Create a basic layer.

Use bitbake-layers <subcommand> --help for assistance on a specific command.

As you can see, the script offers capabilities beyond layer creation. It enables layer addition to the configuration, removal of layers from the build configuration, display of currently added layers, and other useful operations. You can also explore features such as listing recipes present in the build or a specific layer.

Let's now examine the layers currently included in our configuration.

$ bitbake-layers show-layers

The output should resemble the following:

NOTE: Your output may vary based on your previous builds. The following is a representative example.

NOTE: Starting bitbake server...

layer	path	priority
meta	/home/shashank/code/yocto/poky/meta	5
meta-poky	/home/shashank/code/yocto/poky/meta-poky	5
meta-yocto-bsp	/home/shashank/code/yocto/poky/meta-yocto-bsp	5

Step 1 – Selecting the Location for Your Layer

Assuming you have already read Part 2 and Part 3, you are familiar with our folder structure. However, if yours differs, simply adjust the following commands to maintain consistency with your paths [38].

yocto/

meta-openembedded

meta-raspberrypi

poky/

bitbake

build

meta

...

Placing your custom layer within the yocto/ folder, alongside other non-poky layers you have added, is generally considered a good practice. Although not mandatory, it is recommended to prefix the layer name with "meta-".

For our demonstration, let's create a layer named "meta-ke" inside the yocto/ folder.

Step 2 – Creating the Layer

Assuming we are currently in the poky/build/ directory, execute the following commands:

```
$ bitbake-layers create-layer ../../meta-ke
```

The folder structure should now resemble the following:

yocto/

meta-openembedded

meta-raspberrypi

meta-ke

poky/

bitbake

build

meta

...

Adding the Layer to the Build Configuration

With the successful creation of our new layer, let's proceed to add it to our configuration. In the future, we will incorporate our own recipes into this layer, allowing us to create customized images. Once again, we will employ the bitbake-layers script for this purpose [38].

```
$ bitbake-layers add-layer ../../meta-ke
```

To confirm the successful addition, run the bitbake-layers script with the "show-layers" argument. Our layer should now be visible in the output.

The output should resemble the following:

```
layer                 path                                      priority
==========================================================================
meta                  /home/shashank/code/yocto/poky/meta       5
meta-poky             /home/shashank/code/yocto/poky/meta-poky  5
meta-yocto-bsp        /home/shashank/code/yocto/poky/meta-yocto-bsp  5
meta-ke               /home/shashank/code/yocto/meta-ke         6
```

Testing the Recipe Within the New Layer

When the bitbake-layers script created our layer, it also generated an example recipe named "example.bb" located at meta-ke/recipes-example/example/.

Let's attempt to build this recipe. A successful build indicates that the layer operations were successful, and we can now utilize this layer for all future image builds!

```
$ bitbake example
```

During the build process, you should observe output similar to the following. If you have previously conducted an image build, this step should be relatively quick.

```
NOTE: Executing Tasks
**********************************************
*  Example recipe created by bitbake-layers          *
**********************************************
```

12.6 Creating your first custom recipe

Let's work on rewriting the text you provided for your book. Here's a revised version:

Chapter 1: The Art of Yocto Recipe Creation

Preparing for Your First Recipe

Are you ready to embark on the journey of creating your very own recipe? Before we dive into the excitement, let's cover the basics of Yocto recipes. In this chapter, we'll explore how to name recipes, how BitBake discovers them, and how to write a basic recipe for a simple use-case. Finally, we'll delve into the fascinating world of recipe applications.

Recipes: An Overview

In the previous section, we learned that Yocto is a highly scalable build system widely used for creating Linux distributions across various devices. At the heart of this system lies BitBake, a powerful scheduler and execution engine. BitBake interprets metadata and performs actions accordingly, making it the workhorse of the entire build process [38].

In Yocto terminology, a BitBake recipe, or simply a recipe, is a form of metadata that guides BitBake in obtaining, configuring, patching, building, and installing individual software components. Drawing an analogy to real-life cooking recipes, a BitBake recipe contains all the essential information, from ingredients to presentation, to bring a software component to life [38].

The Yocto documentation outlines a standard process for creating BitBake recipes, as depicted in the image below. However, it's important to note that not all recipes follow these exact steps. Some steps may be skipped depending on the specific requirements. For instance, certain recipes might not require patches or even compilation!

[Image: Process of creating a BitBake recipe]

Naming Conventions for BitBake Recipes

Naming a BitBake recipe is a straightforward task. The recipe follows a simple structure:

basename_version.bb

The basename represents the core name of the recipe. It should not include any reserved prefixes or suffixes like "cross," "native," or "lib." However, if the basename itself contains such words, they can be used. For example, if you have a magical recipe to delight users, feel free to name it "turnusercross_0.1.bb." The idea is to avoid tying the recipe to a specific architecture or type, as Yocto can handle these distinctions automatically (more on this in a later section [38].

The version can be a dot-separated string, typically following a major.minor or major.minor.patch notation. Here are a couple of acceptable examples:

- somerecipe_1.2.bb

- anotherrecipe_1.2.1.bb

Recipe Placement in Yocto Layers

BitBake recipes are always located within Yocto layers. To make a recipe discoverable by BitBake, the corresponding layer must be added to the current build configuration. Need a refresher on adding a layer to your build? Refer to Part 5 of this series!

Once you've added the layer containing your recipe to the build configuration, let's explore how BitBake finds it.

To understand this, let's examine the conf/layer.conf file in the layer we created in the previous section [38].

```
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
        ${LAYERDIR}/recipes-*/*/*.bbappend"
.....
.....
```

The relevant configurations here are BBPATH and BBFILES.

In the auto-generated layer.conf:

- The BBPATH variable functions similarly to the PATH variable in our system. It helps BitBake locate bbclass and configuration files [38].

- The BBFILES variable is a space-separated list

of recipe files used by BitBake for software builds. Importantly, it allows the usage of wildcards (*) in the file path, enabling you to specify generic paths. The syntax resembles Python's glob functionality.

Based on this information, it's clear that as long as your recipe is located in a folder inside the layer named "recipes-", BitBake will successfully discover it[38].

A Skeleton BitBake Recipe

To kickstart your own custom recipe development, you can use the following skeleton recipe. This template is directly sourced from Yocto's manual. Note that not all of these sections are mandatory; consider them as recommendations [38].

```
DESCRIPTION = "This is a short description of the recipe"
HOMEPAGE = "Specify your homepage here"
LICENSE = "Type of license (e.g., GPLv2, MIT)"
SECTION = "Useful for package management (can be omitted)"
DEPENDS = "Specify dependencies for your recipe"
LIC_FILES_CHKSUM = "md5sum or sha256sum of the license"

SRC_URI = "Specify the source code location"
```

Pro Tip: Save this skeleton recipe for future reference!

Writing a Recipe for a Simple "Hello World" Printer Application

In the previous section, we created a custom layer with the following structure:

```
meta-ke
  conf/
  COPYING.MIT
  README
  recipes-example/
```

Let's now create a new folder named "recipes-ke" to house our custom recipe.

This adheres to the requirements set by the BBFILES setting in conf/layer.conf. The updated folder structure should look like this:

```
meta-ke
  conf/
  COPYING.MIT
  README
  recipes-example/
  recipes-ke/
```

Next, we'll write a simple "Hello World" C code for a printer application and name the file "hello-world-local.c."

```c
#include <stdio.h>

int main(void) {
   printf("Hello, World!\r\n");
   return 0;
}
```

Since our recipe uses a locally available source tree, we need to create a folder called "files" within the same directory as the recipe (*.bb) file. To maintain organization, let's create a folder named "hwlocal" inside "recipes-ke" and place the "files" folder, along with the source code, inside it. Finally, we'll create a recipe named "hwlocal_0.1.bb" within the "hwlocal" folder.

Now, the updated "meta-ke" folder structure appears as follows:

```
meta-ke
  conf/
  COPYING.MIT
  README
  recipes-example/
  recipes-ke/
    hwlocal/
      files/
```

```
        hello-world-local.c
      hwlocal_0.1.bb
```

Let's use the skeleton recipe as a starting point and write "hwlocal_0.1.bb" to understand what each section entails and why:

```
DESCRIPTION = "This is a simple Hello World recipe that uses a local source file"
HOMEPAGE = "https://kickstartembedded.com"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://hello-world-local.c"

S = "${WORKDIR}"

do_compile() {
    ${CC} ${LDFLAGS} hello-world-local.c -o hello-world-local
}

do_install() {
    install -d ${D}${bindir}
    install -m 0755 hello-world-local ${D}${bindir}
}
```

```

Let's break down the sections and understand their purposes:

DESCRIPTION:

This

section provides a simple description of what the recipe does.

HOMEPAGE:

Usually, this field points to a web address where others can find more information about the recipe, the author, or the owner.

LICENSE:

This field is crucial for a successful build. It informs BitBake about the type of license associated with the source code used in the recipe. Poky comes with a collection of commonly used licenses, which can be found in your Poky installation under "poky/meta/files/common-licenses."

LIC_FILES_CHKSUM:

This field is equally important for a successful build. It specifies the location of the license file and its checksum. During the Quality Assurance (QA) stage, BitBake calculates the checksum and compares it to the value specified here. If they don't match, the build fails. In this recipe, we refer to the license present in the Poky installation using the BitBake environment variable "COMMON_LICENSE_DIR."

SRC_URI:

This field tells BitBake where to find the source code. In this case, we're pointing BitBake to a locally available file called "hello-world-local.c." By default,
```

BitBake looks for this file in the "files" folder. Note that we use the "file://" prefix to indicate a local resource[38].

Now, let's move on to some critical configurations:

S:

This variable specifies the location in the build directory where the unpacked recipe source code resides. When dealing with tarballs, you typically don't need to specify this explicitly unless the extracted folder's name differs from "<recipe name>-<version name>". We'll explore such cases in the next section. In this recipe, since we're working with a local source tree, we need to explicitly specify the location using the environment variable "WORKDIR," which represents the active working directory for BitBake[38].

do_compile():

This section informs BitBake about the command to execute for compiling the source tree. In this case, it's a relatively simple one-line C command. However, as the source tree grows, the compilation process can become more complex. Note how the compiler name and flags are passed as "CC" and "LDFLAGS," respectively. These are environment variables populated by BitBake[38].

do_install():

This section specifies where to install the resulting binary output after compilation. The output is a binary named "hello-world-local." In this case, we tell BitBake to install the binary into the "/bin" folder of the target machine's generated image. The environment variable "bindir" expands as "/bin," and the environment variable "D" represents the path of the target machine's image[38].

Pro Tip: To explore all the environment variables for your build configuration, you can execute the following command after sourcing the "oe-init-build-env" script:

```
```

```
$ bitbake -e
```

Review the output to see the values of "WORKDIR," "S," "D," "COMMON_LICENSE_DIR," "AVAILABLE_LICENSES," "MACHINE," and more.

In the next section , we'll learn how to specify sources located in a git repository, remote tarball, or local tarball[38].

12.7 Customizing images by adding your recipes

In this section, let's discuss how to incorporate the work done in parts 5 to 8 and modify an existing image by integrating our recipes. We will explore various methods to achieve this, focusing on the most commonly used techniques[38].

Strategy for Customizing an Image

Developing a custom image follows a straightforward strategy. You take an existing image and add your unique touch to it, like a secret sauce. Within poky, open-embedded, and the meta-layers provided by your machine providers, you'll find a range of basic image recipes. Make good use of them! To summarize, visualize the strategy as follows:

A simplified visualization of our strategy

This approach saves time, ensures safety, and offers future-proofing compared to building an image from scratch. Creating an image from scratch can introduce various issues. For instance, you might overlook essential packages, leading to

build failures, or worse, the system may encounter critical errors during boot-up[38].

Bottomline: Whenever possible, avoid creating a new image; instead, customize an existing one!

There are multiple methods to add your recipes to an existing image or build configuration. Let's explore the most common approaches, starting with the easiest.

- **[Easiest] Method 1 – Customizing the Build Configuration (Editing local.conf File)**

This is perhaps the simplest way to customize an image.

To customize the build configuration, you can edit the local.conf file located in the conf/ directory of the build directory. However, please note that this method only allows you to add packages and is not as flexible as creating a fully customized image. Changes made in local.conf affect all builds equally, impacting all images. If you want the additional packages to be part of a specific image only, you must explicitly specify the image[38].

To add a custom package to your image using the local.conf file, you can utilize the IMAGE_INSTALL variable with the _append operator. For example, to include the recipe created in our previous section in the build, add the following line to the local.conf file:

IMAGE_INSTALL_append = " hwgit"

Note the syntax: there should be a space between the quote and the package name, in this case, "hwgit." This space is necessary because the _append operator does not add the space automatically[38].

Furthermore, it's recommended to use _append instead of the += operator (as advised by some online resources) to avoid ordering issues. The basic usage demonstrated above affects all images. However, you can extend the syntax to apply the variable to a specific image only. For instance, if you want to add a package to the core-image-minimal image exclusively, use the following:

IMAGE_INSTALL_append_pn-core-image-minimal = " hwgit"

The above change to local.conf will add "hwgit" to the core-image-minimal image only. Additionally, you can use a similar approach with the CORE_IMAGE_EXTRA_INSTALL variable to add packages specifically to core-image-* images.

- **[Easier] Method 2 – Creating an Image Recipe based on an Existing Recipe**

This is another convenient and clean method for building a custom image.

The process is simple: create a new image recipe (.bb file), specify the base image you want to build upon, and then define the new packages to be added.

Let's consider our meta-ke layer used in our previous sections (here, here, here, and here). Inside the meta-ke folder, create a new directory named "images" and within it, create a new recipe called "ke-core-image-minimal.bb" with the following content:

We base this recipe on core-image-minimal core image require recipes-core/images/core-image-minimal.bb

Adding a description is optional but often helps the user understand the purpose of the image

DESCRIPTION = "This is a customized version of the core-image-minimal image available in poky - this contains dev packages and installs the custom hello-world recipes present in meta-ke."

We specify the additional recipes to be added to the build here

IMAGE_INSTALL += "hwlocal hwgit"

Note: In this example, we directly use the IMAGE_INSTALL variable without any ordering risk. Additionally, we haven't included whitespace in this case,

unlike the usage of IMAGE_INSTALL_append mentioned earlier, as the += operation automatically takes care of adding the whitespace for us.

That's it! You can now build your custom "ke-core-image-minimal.bb" image.

- **[Easy] Method 3 – Customizing the Image Recipe by Appending to It**

There might be situations where modifying an existing core image is necessary but creating a custom .bb recipe file isn't feasible due to reusability or legacy reasons. In such cases, instead of creating a brand new .bb file, you can create a .bbappend file with the same name as the core recipe you want to modify or extend. Bitbake provides bbappend files as a means to append metadata to an existing configuration.

For example, let's say we cannot create "ke-core-image-minimal.bb" as described in method 2. However, we still need to include "hwgit" and "hwlocal" in our core-image-minimal builds. To achieve this, create a "core-image-minimal.bbappend" file inside the meta-ke/images folder with the following content:

IMAGE_INSTALL += "hwlocal hwgit"

Note: This is a simple example, but .bbappend files can become more complex when building elaborate images. The key to keeping things simple is to choose a base image that closely aligns with your desired custom image[38].

Now, you can generate your custom image without changing the name of the base recipe while having your custom packages included in the image [38].

12.8 Summary

In this section, we explored three different ways to quickly add custom packages to your image build. While there are additional approaches available, the methods discussed here are commonly used, easy to implement, and align with our core strategy of leveraging the layered nature of Yocto.

References

[1] Ronald Jurgen,” V2V/V2I Communications for Improved Road Safety and Efficiency”, SAE International, Warrendale, 2012.

[2] Cindy G. Goldsberry,” ZFactor Sales Accelerator V2V: From Vendor to Value Creator”, CreateSpace Independent Publishing Platform ,2012.

[3] Derek Molloy,”Exploring Raspberry Pi”, John Wiley & Sons, Indianapolis, IN , 2016.

[4] Kristin Fontichiaro, Charles R ,” Raspberry Pi”, Cherry Lake Publishing, 2014.

[5] Luc Volders,” ESP32 Simplified”, Lulu Press , Inc, 2020.

[6] Derek Esp,” Adventure in Education: The Autobiography of an Accidental Career in Education”, Troubador Limited, Inc , 2019.

[7] Lamont Poulin,” Raspberry Pi Setup Guide : How to Use Your Raspberry Pi and Python Programming from Scratch: Raspberry Pi Setup Ssh”, Independently Published,2021.

[8] Tamera Kesley,” Raspberry Pi Step by Step Guide : Using Your Raspberry Pi: Face Recognition Using Raspberry Pi Synopsis”, Independently Published, 2021.

[9] Robert Dunne,” Assembly Language Using the Raspberry Pi: A Hardware Software Bridge”, Gaul Communications, 2017.

[10] Michael G. Robinson, Jianmin Chen, Gary D. Sharp ,"Polarization Engineering for LCD Projection",John Wiley & Sons, Ltd ,Chichester, UK ,2005

[11] Brown A. E, E. G. Richardson ,"Ultrasonic Physics",Elsevier,2013

[12] Chang-liang Xia," Permanent Magnet Brushless DC Motor Drives and Controls", Wiley & Sons, Incorporated, John,2012

[13] "Complete Guide for HMC5883L Magnetometer /Digital Compass" ,Website: ElectronicWings - Hardware Developers Community

[14] Tracey Scott Wilson ,"Buzzer", Dramatist's Play Service,2016

[15] Luc Volders, "ESP32 Simplified" ,Lulu Press,2020.

[16] DroneBot Workshop, "ESP NOW - Peer to Peer ESP32 Communications", DroneBot Workshop,2022, website.

[17] Neil Cameron, "ESP32 Formats and Communication", Apress, Berkeley,CA,2023.

[18] Juliet Zhu ,"DsPIC33E/PIC24E FRM - UART", Microchip Technology Incorporated,2014.

[19] Linda Pierce, "TB3200 - UART with DALI Protocol Technical Brief", Microchip Technology Incorporated,2018.

[20] Eric Peña,Mary Grace Legaspi,"UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter",Eric Peña,Mary Grace Legaspi,2022 ,Website.

[21] BPB Publications ,” Bluetooth Basics”, Independent Publishers Group, 2001.

[22] Erica White, “Bluetooth: The Everything Guide to Bluetooth Technology”, Lulu Press, Inc, 2015.

[23] Kristin Fontichiaro, Charles R Severance, "Raspberry Pi”, Cherry Lake Publishing,2013.

[24] High Level Design (HLD).

[25] Hallinan, Christopher. Embedded Linux primer: a practical real-world approach.
[26] Pearson Education India, 2011.Raghavan, Pichai, Amol Lad, and Sriram Neelakandan. Embedded Linux system design and development. Auerbach Publications, 2005.
[27] Abbott, Doug. Linux for embedded and real-time applications. Elsevier, 2011.
[28] Simmonds, Chris. Mastering Embedded Linux Programming. Packt Publishing Ltd, 2017. Bi, Chun-yue, Yun-peng Liu, and Ren-fang Wang. "Research of key technologies for embedded Linux based on ARM." 2010 International Conference on Computer Application and System Modeling (ICCASM 2010). Vol. 8. IEEE, 2010.
[29] Vaduva, Alexandru, Alex Gonzalez, and Chris Simmonds. Linux: Embedded Development. Packt Publishing Ltd, 2016.
[30] Hallinan, Christopher. "Reducing boot time in embedded linux systems." LINUX journal 2009.188 (2009): 4.
[31] Wang, Ya-Jun. "Research and realization of the mechanism of embedded linux kernel semaphore." 2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE). Vol. 1. IEEE, 2010.
[32] Molloy, Derek. Exploring Raspberry Pi: interfacing to the real world with embedded Linux. John Wiley & Sons, 2016.
[33] Simmonds, Chris. Mastering embedded Linux programming. Packt Publishing Ltd, 2015.
[34] Salvador, Otavio, and Daiane Angolini. Embedded Linux Development with Yocto Project. Packt Publishing Ltd, 2014.
[35] Introduction to Yocto Project : > Welcome to the Yocto Project Documentation.
[36] Yocto Project Overview :> Yocto Project Reference Manual
[37] Yocto project Quick Build:>Yocto Project Quick Build

الملخص العربي

تعتبر تقنيه نظم الاتصالات "V2V "من التقنيات المهمة في مجال السيارات الذكية الذاتية القيادة، حيث تسمح بتبادل المعلومات بين المركبات المتواجدة على الطريق. وتتيح هذه التقنية للسيارات التواصل مع بعضها البعض وتحديد مواقعها وسرعتها واتجاهها، مما يساعد على الحفاظ على الارواح وتقليل عدد الحوادث المرورية حيث يمكن للمركبات أن تتوقع وتتفاعل مع مواقف القيادة المتغيرة ثم تقوم على الفور بتحذير السائقين برسائل تحذير طارئة. إذا لم يستجب السائق لرسالة التنبيهات، يمكن للمركبة أن توقف نفسها بأمان تجنبا للإصطدام.

لذلك، في هذا المشروع يتم تنفيذ نظام اتصالات في الوقت الفعلي "Real Time" من مركبة إلى مركبة عبر شبكة WIFI محلية بين السيارات حيث يتم بث بيانات السيارة والحالة لتجنب الحوادث, وقد تم حصر ذلك في خمس تطبيقات رئيسية وهم:

1. Blind Spot Warning (BSW) : هو نظام التحذير من النقطة العمياء, حيث يستخدم لتحذير السائق عندما يكون هناك مركبة في المنطقة التي لا يمكن للسائق رؤيتها بوضوح باستخدام المرآة الخلفية والجانبية فعندما يتم رصد مركبة في هذه المنطقة، يقوم النظام بإرسال إشارة تحذيرية إلى السائق، عادة عن طريق إشارة صوتية أو رسالة تظهر على شاشة العرض في لوحة القيادة.
2. Emergency Electronic Brake Light (EEBL) : وهو نظام يستخدم في السيارات لتحذير السائقين عند حدوث حالة طوارئ تتطلب إيقاف السيارة بشكل مفاجئ وعاجل، مثل حالة الفرملة الطارئة.
3. Forward Collision Warning (FCW) : ويعمل هذا النظام عن طريق استخدام مجموعة من الحساسات والكاميرات المثبتة في السيارة لرصد المركبات الأمامية، وعندما يتم رصد مركبة أمامية تتجه باتجاه السيارة بشكل محتمل للتصادم، يقوم النظام بإصدار إشارة تحذيرية للسائق، عادة عن طريق إشارة صوتية أو رسالة تظهر على شاشة العرض في لوحة القيادة.
4. Intersection Movement Assist (IMA) : هو نظام يستخدم لتحسين سلامة القيادة في تقاطعات الطرق المزدحمة. يعمل النظام عن طريق استخدام مجموعة من الحساسات المثبتة في السيارة لرصد المركبات وعند اقتراب السيارة من تقاطع، يقوم النظام بتحليل البيانات المستشعرة وإصدار إشارة تحذيرية للسائق عندما يكون هناك خطر اصطدام محتمل بالمركبات المتجهة في الاتجاه المعاكس
5. Don't Pass Warning (DNPW) : هو نظام يستخدم في السيارات لتحذير السائق عندما يكون من المحتمل أن يكون هناك خطر في تجاوز المركبات الأمامية بشكل خطر أو غير آمن. عندما يحاول السائق تجاوز مركبة ما بشكل خطر، يقوم النظام بإصدار إشارة تحذيرية للسائق، عادة عن طريق إشارة صوتية أو رسالة تظهر على شاشة العرض في لوحة القيادة.

Valeo

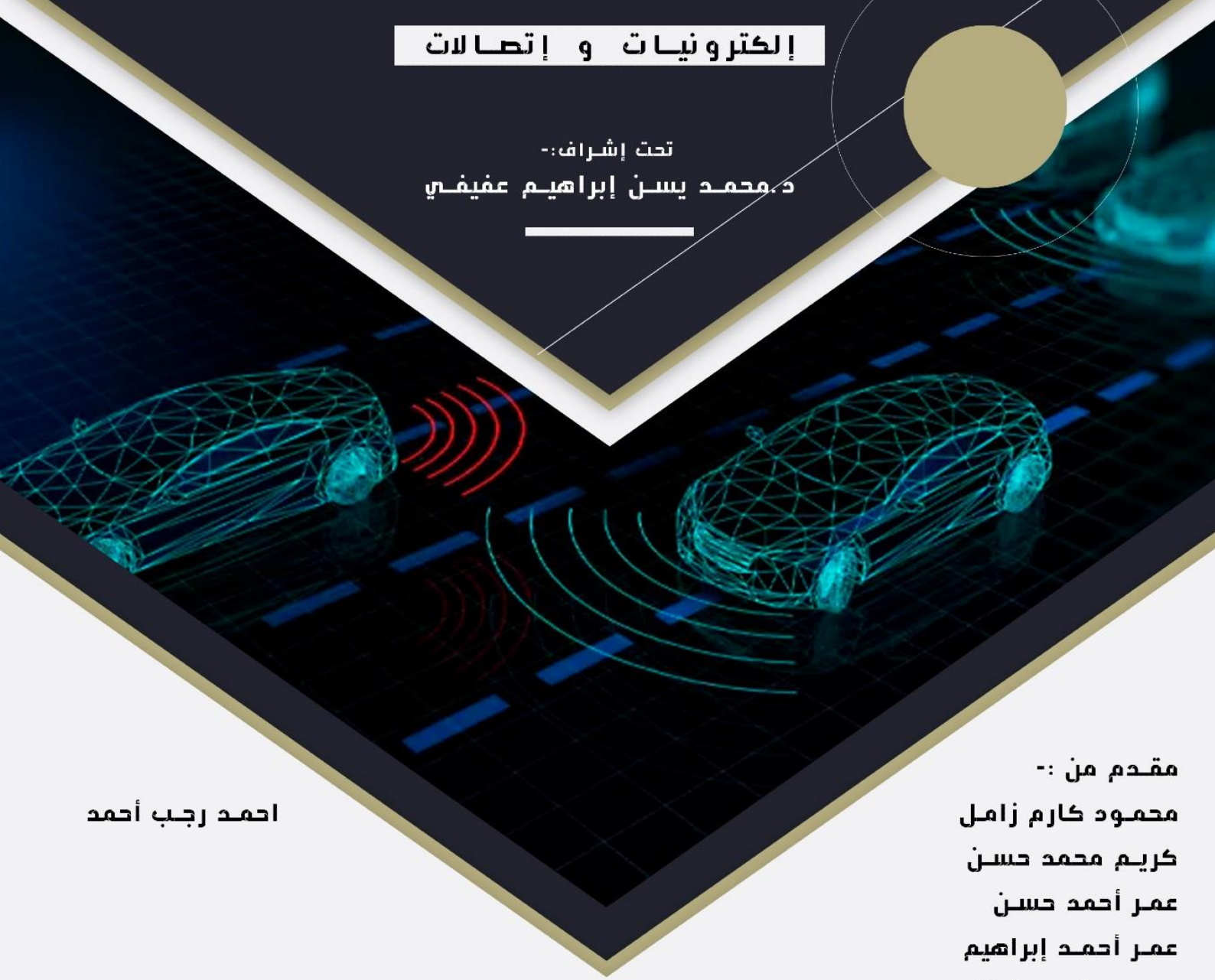
جامعة الأزهر
كلية الهندسة
قسم الإلكترونيات والإتصالات

مشروع تخرج بكالوريوس

التواصل بين المركبات لتفادي الاصطدامات

إلكترونيات و إتصالات

تحت إشراف:-
د.محمد يسن إبراهيم عفيفي



مقدم من :-
محمود كارم زامل
كريم محمد حسن
عمر أحمد حسن
عمر أحمد إبراهيم
محمد ياسر كمال
محمد جميل الزند

احمد رجب أحمد